VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV MECHANIKY TĚLES, MECHATRONIKY A BIOMECHANIKY

INSTITUTE OF SOLID MECHANICS, MECHATRONICS AND BIOMECHANICS

VYUŽITÍ NÁSTROJE MATLAB CODER PRO AUTOMATICKÉ GENEROVÁNÍ C KÓDU PRO MIKROKONTROLÉRY DSPIC

APPLICATION OF MATLAB CODER FOR AUTOMATIC GENERATION OF C CODE FOR DSPIC MICROCONTROLLERS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — **Bc. Tomáš Mácha**

AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Robert Grepl, Ph.D.**

SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání diplomové práce

Ústav:	Ústav mechaniky těles, mechatroniky a biomechaniky
Student:	**Bc. Tomáš Mácha**
Studijní program:	Aplikované vědy v inženýrství
Studijní obor:	Mechatronika
Vedoucí práce:	**doc. Ing. Robert Grepl, Ph.D.**
Akademický rok:	2018/19

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma diplomové práce:

Využití nástroje MATLAB Coder pro automatické generování C kódu pro mikrokontroléry dsPIC

Stručná charakteristika problematiky úkolu:

Automatické generování kódu z prostředí MATLAB/Simulink pro vestavěné (embedded) systémy je běžnou součástí průmyslové praxe i výuky na univerzitách. Převažuje však využití Simulinku a nástrojů Simulink Coder a Embedded Coder společně s balíčkem specifickým pro danou platformu (RPi, Arduino, LEGO, dsPIC a další). V případě generování kódu z jazyka MATLAB pomocí MATLAB Coder převažuje využití v rozsáhlejších projektech a kód je pak např. spouštěn v prostředí specializovaných RTOS.

Tato práce se bude zabývat vývojem prostředí pro výukové a případně jednoduché průmyslově použitelné aplikace, kdy dostačuje použití cenově dostupného mikrokontroléru s poměrně jednoduchým časováním jednotlivých funkcí. Obsluha periferií a časování bude naprogramováno pevně pro vybraný mikrokontrolér, z prostředí MATLABu se bude generovat funkční kód, který se automaticky propojí s rámcem aplikace.

Cíle diplomové práce:

1) Seznamte se s problematikou automatického generování C kódu pro vestavěné aplikace, konkrétně s nástroji MATLAB Coder, Simulink Coder, Embedded Coder, Blockset for Microchip dsPIC Microcontrollers.
2) Pro vybrané periferie (např. DIN, DOUT, PWM, ADC) a zvolený způsob časování aplikace vytvořte v jazyce C rámec aplikace, do kterého bude možné vkládat automaticky vygenerované funkce z MATLABu.
3) V prostředí MATLAB GUI vytvořte aplikaci, která umožní konfiguraci periferií a generovaného kódu a usnadní uživatelskou práci s vytvořeným nástrojem.
4) Zpracujte sérii demonstračních úloh (např. filtrace ADC signálu, řízení DC motoru apod.) na kterých ukažte vlastnosti a ověřte limity vytvořeného řešení.

Seznam doporučené literatury:

dokumentace k MATLAB Coder toolboxu

Herout, P.: Učebnice jazyka C

Valášek, M.: Mechatronika, Vydavatelství ČVUT 1995

Dušek, F.: Matlab a Simulink, skriptum ČVUT

Váňa, V.: Mikrokontroléry ATMEL AVR - assembler, Nakladatelství BEN, 2003

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2018/19

V Brně, dne

L. S.

prof. Ing. Jindřich Petruška, CSc.
ředitel ústavu

doc. Ing. Jaroslav Katolický, Ph.D.
děkan fakulty

Abstract

Generating C code from MATLAB can be a powerful tool for a wide range of engineering problems. Especially in cases where MATLAB excels, including matrix operations and some of the built-in functions, code conversion may represent a significant assistance in their development. The aim of this master's thesis is to explore the possibilities of generating C code using MATLAB Coder App. As the documentation for MATLAB Coder was not found sufficient a theoretical basis for the use of this tool was introduced along with simple examples of possible applications. Next, code generation for controlling a laboratory DC motor using a microcontroller dsPIC was made automatic and controlled directly from a single MATLAB script. Individual steps were described to supplement existing documentation and to allow for an easy use. Finally, selected functions were tested for code generation. As a result, certain limitations can be outlined, as well as suitability of both the created tool and the MATLAB Coder App.

Key words

MATLAB, MATLAB Coder App, Command Line, MPLAB Device Blocks for Simulink, Double drive, dsPIC33FJ128MC804, Arduino.

Abstrakt

Generování kódu C z MATLABu může představovat mocný nástroj u široké škály inženýrských problémů. Zejména pak u operací ve kterých MATLAB vyniká, zahrnujících operace s maticemi a některé z vestavěných funkcí, může převod kódu představovat výrazné urychlení vývoje jejich aplikací. Cílem této diplomové práce je prozkoumat možnosti generování kódu C pomocí balíku MATLAB Coder. Jelikož byla dokumentace k MATLAB Coderu shledána jako nedostatečná, byl do práce začleněn také teoretický základ pro použití tohoto balíku a to spolu s jednoduchými ukázkami možných aplikací. Dále byl navržen a popsán nástroj pro automatcké generování kódu pro řízení laboratorního modelu DC motoru využívající mikrokontrolér dsPIC. Jednotlivé kroky byly popsány a doplňují tak stávající dokumentaci. Závěrem práce je testováno generování kódu také pro vybrané funkce. Díky tomu mohou být nastíněna jistá omezení a obecně také vhodnost jak vytvořeného nástroje, tak také balíku MATLAB Coder.

Klíčová slova

MATLAB, MATLAB Coder App, Příkazová řádka, MPLAB Device Blocks for Simulink, Double drive, dsPIC33FJ128MC804, Arduino.

Rozšířený Abstrakt

Generování kódu z prostředí MATLAB nebo Simulink pro vestavěné (embedded) systémy lze již mnoho let považovat za běžnou praxi v průmyslových odvětvích. V roce 2004 byl poprvé představen „Embedded MATLAB Function Block“ v Simulinku. Jen o několik let později vyšel také Simulink Coder (dříve známý jako Real-Time Workshop), umožňující generování C kódu ze Simulinku. MATLAB Coder byl uveden v roce 2011 a výrazně rozšířil stávající technologie.

Od té doby se průmyslové procesy a také systémy, které je ovládají, staly složitějšími a objevily se nové technologické výzvy. MATLAB a Simulink, které lze považovat za široce používané nástroje, často v těchto procesech hrají důležitou roli. Je-li součástí i jakákoliv aplikace programovacího jazyka C/C++, pak se právě možnost jeho generování z MATLABu stává značnou výhodou. Zejména je-li generování kódu prováděno automaticky, může tato technologie představovat například značnou úsporu času.

MATLAB a Simulink jsou, mimo jiné, také často prvními nástroji, které se vyučují na technických univerzitách a oproti C/C++ jsou relativně snadno aplikovatelné. Velký rozdíl v náročnosti aplikace se projevuje zejména v případech, kde MATLAB exceluje (např. operace s maticemi, ...). MATLAB, nejen v těchto případech, obsahuje spoustu již vestavěncých funkcí oproti jazyku C/C++, což umožňuje provádění složitých výpočtů s minimem napsaného kódu. Zejména tato odvětví pak mohou s výhodou využít možnost automatického generování C kódu, respektive převod funkcí napsaných v MATLABu na funkce jazyka C/C++.

Jakkoli se může zdát MATLAB Coder užitečný, v současné době neexistuje mnoho článků či publikací popisujících jeho použití v reálných aplikacích. Způsob popisu takového použití (např. v uživatelské příručce MATLAB Coder) je také často do jisté míry omezený co se týče rozsahu či praktických rad. Cílem práce je prozkoumat možnosti nástroje MATLAB Coder a vyvinout rozhraní pro jednoduchou aplikaci s dalším využitím. Jako jedna z vhodných aplikací bylo zvoleno řízení kartáčovaného stejnosměrného motoru, který je součastí laboratorního modelu „double drive“. V tomto případě postačí použití cenově dostupného mikrokontroléru dspic33FJ128MC804 s poměrně jednoduchým časováním jednotlivých funkcí. Obsluha periferií a časování bude naprogramováno pevně a z prostředí MATLAB se bude generovat funkční kód, který se automaticky propojí s rámcem aplikace. Proto bude možné změnit funkční část pouze úpravou funkce s předdefinovanými vstupy/výstupy MATLABu.

V reakci na nedostatečnou dokumentaci takového použití bude poskytnut detailní popis průběhu generování kódu. To by mělo zejména usnadnit možnost efektivního použití nástoje MATLAB Coder pro budoucí uživatele. Diskutovány budou dvě metody. Nejprve bude pro generování kódu C použita aplikace s grafickým rozhraním „MATLAB Coder App“ a jako druhý bude použit skript s příkazy nejen pro převod kódu, ale také pro spojení s rámcem projektu, kompilaci a nahrání firmware do mikrokontroléru.

Práce v úvodu obsahuje popis použitého hardwaru. Pro řízení DC motoru v rámci Double Drivu byl použit PIC Edu Kit se zmíněným mikrokontrolérem dsPIC. Ovládání je řešeno pomocí čtyř tlačítek, kde dvě řídí směr otáčení a další dvě slouží k zapnutí/vypnutí brzdy. Pro řízení otáček je využit signál z potenciometru a pro indikaci stavu DC motoru LED kontrolky. Jako programátor je použit PICkit 3. Samotný Double Drive obsahuje H-můstek, jehož funkce je v práci blíže popsána.

Úvodní kapitola o samotném MATLAB Coderu začíná popisem historického vývoje generování kódu C z MATLABu/Simulinku. Dále je stručně popsáno teoretické minimum, které bylo v průběhu řešení projektu shledáno užitečným. To obsahuje například výčet podporovaných datových typů, užitečné příkazy pro nastavení a ovládání MATLAB Coderu pomocí příkazové řádky atd.

Způsob převodu kódu do jazyka C je uveden testováním možností generování kódu ze Simulinku, a to jak pro zmíněný mikrokontrolér, tak pro Arduino. Vygenerovaný kód byl prozkoumán a následně z něj byla čerpána inspirace pro vytvoření rámce celého projektu pro řízení Double Drivu. Rámec projektu, který obsahuje především konfiguraci časovačů, přerušení a další nastavení mikrokontroléru, je naprogramován pevně v jazyce C. Generované funkce z MATLABu se po jejich vytvoření s tímto rámcem automaticky propojí a následně jsou volány právě pomocí přerušení, a to každá s rozdílnou frekvencí. První z funkcí je jednoduchý filtr používající tzv. klouzavý průměr (Moving Average Filter), který filtruje signál z potenciometru s využitím ADC. Tento signál je následně využit druhou vygenerovanou funkcí nazvanou `step100`, která běží na frekvenci 100Hz a stará se o chod motoru (kontrola stisku tlačítek, rychlost a směr otáčení, brzda on/off).

Využití grafického rozhraní balíku MATLAB Coder bylo krok po kroku popsáno pro generování funkce `step100`. V rámci urychlení převodu kódu z MATLABu lze s výhodou využít takzvanou testovací funkci, která volá funkci pro převod s již nadefinovanými datovými typy, které se v aplikaci MATLAB Coder uloží a nemusí se tak zadávat ručně. Následuje vždy automatická kontrola přítomnosti nepodporovaných funkcí a syntaxe, pro kterou se z konvertované funkce vygeneruje a otestuje MEX soubor. V posledním kroku se zvolí požadovaný vygenerovaný jazyk (C nebo C++) a také upřesní finální nastavení.

Stejných výsledků jako při použití grafického rozhraní balíku MATLAB Coder bylo dosaženo také s využitím příkazové řádky. Tento způsob přináší řadu výhod, neboť veškerá funkcionalita je napsána v jediném skriptu a veškeré změny v nastavení lze provádět velice rychle. Další z významných výhod je, že skript může obsahovat příkazy pro převod hned několika funkcí jazyka MATLAB do jazyka C. Díky charakteru vytvořeného skriptu bylo také možné vytvořit vlastní grafické uživatelského rozhraní (GUI), které umožňuje jednoduché ovládání vytvořeného nástroje.

V závěru práce jsou zhodnoceny také výstupy testování převodu do jazyka C pro vybrané funkce z MATLABu. Jako příklad lze uvést testování filtru, generování náhodných čísel, operace s maticemi, goniometrické funkce, rychlá Fourierova transformace a v závěru také testování generování objektů. Důraz je kladen mimo jiné na popis způsobu testování,

kdy je pro určitou funkci v jazyce C vytvořen program v Simulinku. Zde se tato funkce volá pomocí bloku „C function call“.

Hlavním výsledkem celé práce je vytvořený nástroj umožňující automatické generování C kódu pro ovládání modelu „Double Drive“. Veškerá funkcionalita je zde ovládána pouze z MATLABu, a to včetně kompilace a nahrání projektu na mikrokontrolér. Nelze však opomenout také hodnotu vytvořeného „návodu“ pro podobné použití v budoucnu, které kromě okomentovaných instrukcí u jednotlivých kroků obsahuje i mnoho odkazů na literaturu (zejména uživatelské příručky a webové stránky MathWorks).

Bibliographic citation

MÁCHA, T. Využití nástroje MATLAB Coder pro automatické generování C kódu pro mikrokontroléry dsPIC. Brno: Vysoké učení technické v Brně, Fakulta strojního inženýrství, 2019. 75 s. Vedoucí diplomové práce doc. Ing. Robert Grepl, Ph.D..

Affirmation

I declare that the master's thesis is the result of my own work and all used sources are duly listed in the bibliography.

Brno 22. 5. 2019 .....

Bc. Tomáš Mácha

Acknowledgement

I would like to express my very profound gratitude to my entire family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of writing this thesis. This accomplishment would not have been possible without them.

I would also like to thank doc. Ing. Robert Grepl, Ph.D. for supervising my master's thesis and also to his colleagues in the Mechatronics laboratory.

To Klára, because no matter the circumstances, she does deserve gratitude.

To Ondra, for being like Samwise and staying with me until the very end.

Finally, special thanks also to the University of Southern Denmark and the Norwegian University of Science and Technology where I spent undoubtedly the best and most beneficial part of my studies.

Bc. Tomáš Mácha

Table of contents

Introduction **19**

1 Objectives **21**
1.1 Generating C code from Simulink for Arduino and dsPIC 21
1.1.1 Generating C code using MPLAB Device Blocks 21
1.1.2 Generating C code for Arduino 22
1.1.3 Summary and takeaway for MATLAB Coder applications 23
1.2 Linking MATLAB Coder with external application 24
1.2.1 System description 24
1.2.2 Role of MATLAB 25
1.2.3 Role of MPLAB 25
1.3 Testing functions for conversion 26

2 Hardware & Configuration **27**
2.1 PIC Edu Kit 27
2.1.1 Buttons 28
2.1.2 LEDs 29
2.1.3 Potentiometer 30
2.1.4 dsPIC Microcontroller 30
2.2 PICkit 3 32
2.3 Arduino 33
2.4 Double Drive 34
2.4.1 H-Bridge 34

3 Introducing MATLAB Coder **37**
3.1 Brief History 38
3.2 Useful Functions 39
3.3 Order of Evaluation in Expressions 40
3.4 Data Types 40
3.5 Calling by Reference 41

4 Integrating MATLAB Coder into Double drive control **43**
4.1 MATLAB Coder App 44
4.1.1 Setup of numeric conversion 44
4.1.2 Defining input types 45
4.1.3 Checking for run-time issues 46
4.1.4 Settings and code generation 47
4.1.5 Reviewing generated code 47
4.2 Code generation using command line 49
4.2.1 Creating configuration object 49
4.2.2 Defining argument types 50
4.2.3 Invoking MATLAB Coder 51
4.3 Building and flashing MPLAB project from MATLAB 52

4.3.1 Building the project 53
4.3.2 Uploading Firmware 54

5 Results and Discussion **55**
5.1 Double drive control 56
5.2 Filtering Signals 58
5.3 Finding Eigenvalues 60
5.4 Generating Objects 62

6 Conclusion **63**

Bibliography **65**

List of used symbols and abbreviations **69**

List of figures **71**

List of tables **73**

Appendices **75**
A Attached files 75

Introduction

Code generation from MATLAB/Simulink for embedded systems has been a common part of industrial practices for many years. In 2004 the Embedded MATLAB Function block in Simulink was introduced for the first time. Only a few years later the Simulink Coder (formerly known as Real-Time Workshop) was presented, enabling the generation of a stand alone C code from Simulink. The release of the MATLAB Coder followed in 2011, highly extending the existing technology.

Since that time the industrial processes, as well as systems that control them, have become more complex and new challenges have emerged. MATLAB/Simulink, being widely used tools, have often played an important role in those processes. If the C/C++ functionality is involved, the possibility to generate a portable and readable code of this kind from MATLAB/Simulink then becomes very useful.

Additionally, MATLAB and Simulink are frequently the first tools taught at technical universities and also relatively easy to learn unlike C/C++. On top of that, there are certainly people having real-life research problems dealing with C/C++, while at the same time being familiar with the mentioned tools. In that case, automatic conversion of the MATLAB function to C/C++ might represent a considerable simplification that can overcome at least part of the mentioned issues.

However useful the MATLAB Coder might seem, there are currently not many publications or reports describing its use in real applications. Additionally, the methods delineating such use (e.g. in MATLAB Coder User's Guide) are often notably limited.

The aim of the thesis is to explore the possibilities of MATLAB Coder and to develop an interface for a simple application with further use. Control of a brushed DC motor was chosen as one of the suitable applications. For this case the use of an affordable microcontroller with simple timing of individual functions will be sufficient. The configuration of peripherals and timing will be programmed as fixed in the C language in separate files. However, the functionality will be programed in MATLAB and subsequently converted to the C code. Then the generated code will be automatically linked with the application. Therefore, it will be possible to change the functionality by only changing the MATLAB function with predefined inputs/outputs.

In response to insufficient documentation of such use, a detailed description of code generation will be provided. It should allow anyone willing to use the MATLAB Coder for a similar purpose to progress quickly as well as to profit from other reference materials. Two methods will mainly be discussed. First the MATLAB Coder App and second, the use of a script with commands to not only convert, but also to build and upload the generated code to a microcontroller.

1 Objectives

In this Chapter, the main objectives will be described. Primarily the way of linking MATLAB Coder with an external application and the role of MATLAB. The idea is to program a fixed environment in C language and then generate the functionality in separate functions from MATLAB. However, to fully understand how to effectively generate usable C code from MATLAB, different code generations will first be tested and the resulting codes examined to obtain some inspiration. The testing part is therefore reported at the very beginning.

1.1 Generating C code from Simulink for Arduino and dsPIC

First an arduino UNO was used with C code generated from MATLAB Simulink using the "Simulink Support Package for Arduino Hardware" [1]. Another code was then generated for a dspic33FJ128MC804 microcontroller[1] using MPLAB Device Blocks for Simulink [2] (eventually see [3] for an old Lubin Kerhuel's webpage - some examples provided there are still useful, even though the blocks are now outdated). Following text compares those two approaches based on a very simple example. The example was the same in both cases and the process of generating the C codes is shortly described as well as the code itself. The main reason for doing this is to inspect the structure of generated codes, which could later be imitated for the use of functions generated by MATLAB Coder. All the codes/files which are used in this Chapter can be found in an attachment described in Appendix A and more information can be obtained by inspecting them.

1.1.1 Generating C code using MPLAB Device Blocks

To generate C code for a microcontroller from Simulink a "Master block" has to be configured at first. However, for such a simple example as shown in Fig. 1.1, only the target has to be set by selecting the microcontroller used in the menu of the block. Next, the compiler is set up and finally the desired functionality is programmed using Simulink blocks.

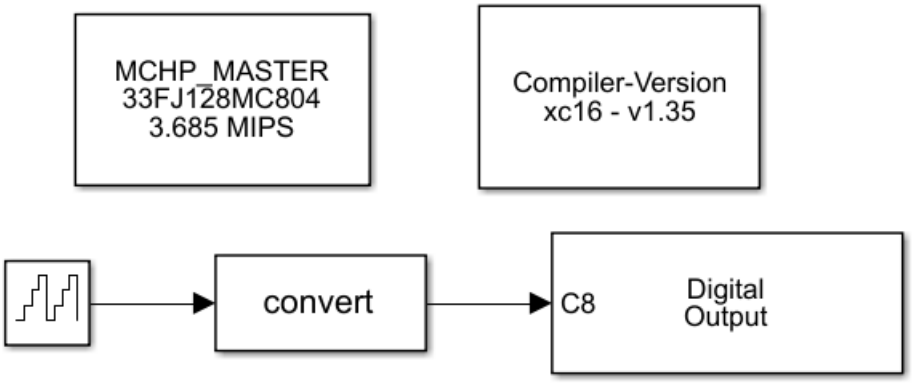

Figure 1.1: Blocks for programming dspic microcontroller - blinking LED.

[1] Thenceforward only refered to as microcontroller or dsPIC

In this example the input to the microcontrollers pin C8 (digital output) consists of a simple free-running counter with specified sample time. This counter overflows back to zero after it has reached the maximum value possible for the specified number of bits (in this case only 2 bits are used). The counter is always initialized to zero. The output is normally an unsigned integer with the specified number of bits and therefore a convert block has to be used to obtain a boolean signal required by the digital output of the microcontroller.

After setting everything up, the C code can be generated. In this case 2 C-files are created:

- `name_main.c`
- `name.c`

where "name" stands for the name of the Simulink file. The first file contains the main function, with setting for pins (I/O, analog/digital), timers (interrupt priority, period) and an endless loop which ensures continuity of operation. The second file contains function `name_step()` which runs as an interrupt service routine (ISR). In this function happens all of the main functionality.

1.1.2 Generating C code for Arduino

Following text applies specifically for Arduino UNO (see section 2.2), therefore, for other types of Arduinos some minor differences may appear. Once again a simple example of blinking LED was made to demonstrate the generated code.

In Fig. 1.2 the configuration can be seen. Input to the Arduino consists of the same free-running 2-bit counter and a convert block to obtain the boolean signal, which is again required by the digital output of Arduino.

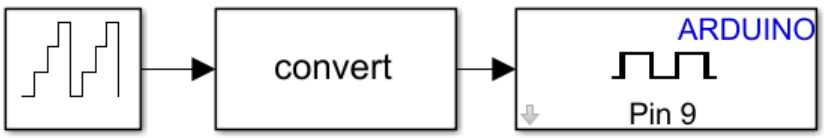

Figure 1.2: Blocks for programming Arduino.

This time there are 3 C-files generated:

- ert_main.c
- name_data.c
- name.c

where ert_main.c contains 2 functions, first is obviously a main function, with all the necessary configuration and once again a while loop to ensure that the program continues to run. This time, also the step function is included there, which is, similarly to previous section, called as an ISR. Also a built-in commands `cli();` and `sei();` are used. `cli();` turns global interrupts off, and `sei();` turns them on. When they are on, any enabled interrupts are followed and make use of the code within an attached ISR function. [4].

1.1.3 Summary and takeaway for MATLAB Coder applications

The biggest takeaway from the previous sections is, that both of the mentioned examples contained their main functionality in a function `step()`, which was called as an interrupt service routine. The same technique will be applied to the functions generated from MATLAB Coder. A variable called `flag` will be made. Whenever an interrupt occurs, this variable will be set to one. The main loop will contain a logical operator, which will then call the generated function. Once the function is executed, `flag` will be set back to zero. This is illustrated in the exemplary code below. As can be seen, the number of generated functions in not limited. Moreover, each can use a specific flag that could be turned on/off with individual frequency. Finally, a configuration function can be generated as well (line 2) which will be outside the while loop and thus will only be executed once.

```
1  int main(void) {
2      generated_config();
3      while(1){
4          if (flag1 == 1) {
5              x = generated_fcn1();
6              flag1 = 0;
7              }
8          if (flag2 == 1) {
9              y = generated_fcn2();
10             flag1 = 0;
11         }
12     }
13 }
```

1.2 Linking MATLAB Coder with external application

To demonstrate the practical use of MATLAB Coder, an external application was chosen to be controlled. Control of a double drive[2] will be described as it offers a wide range of possible applications for the future. Additional functions will be tested and described in section 1.3.

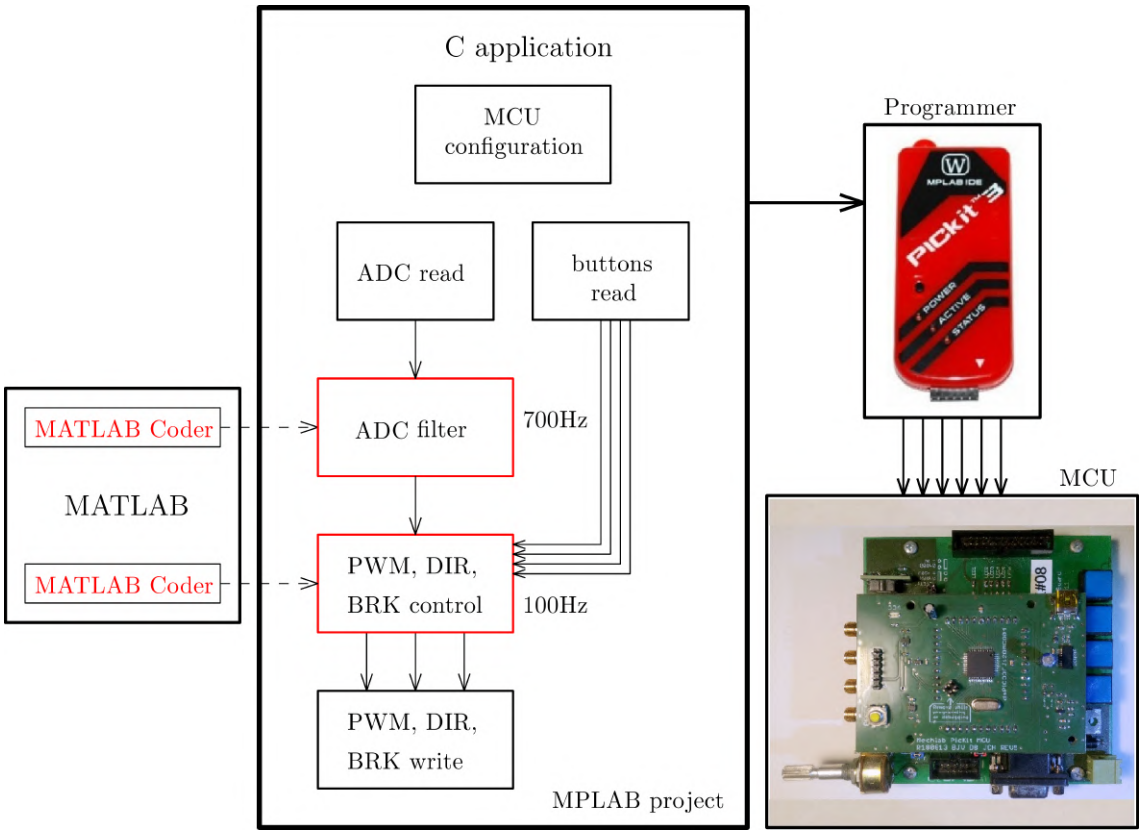

Figure 1.3: Initial plan for linking MATLAB Coder with MPLAB project for a specific example of Double drive control.

1.2.1 System description

The main idea of how MATLAB Coder is intended to be used in double drive control is illustrated in Fig. 1.3. In red are highlighted sections which are directly linked to MATLAB Coder or generated functions. Those are to be generated and automatically connected with the rest of the project from MPLAB and their functionality will be possible to change through MATLAB only. As can be seen, the MCU configuration and some of the other functionalities are planned to be programmed only in MPLAB and MATLAB will not have access to them. Such part will be considered as fixed for the specific application. This approach offers the possibility of omitting tasks like, e.g. peripheral configuration and

[2]laboratory model with two brushed DC motors, see section 2.4 for more information.

therefore lets the user to concentrate only on the math part (control, filtering, logic, ...) of the MATLAB functions. As everything will be controlled from MATLAB, user will not be expected to have extensive knowledge about dsPIC programming.

Used hardware (Double drive, dsPIC microcontroller, PICkit, etc.) will be described in Chapter 2.

1.2.2 Role of MATLAB

The main role of MATLAB will be to generate control functions for double drive. In the following sections, the emphasis will be mostly on describing the process of code generation as converting MATLAB code to C is the main topic of this thesis.

In section 1.1, C code from Simulink was generated and tested. It was observed that the control part of the generated code was called in a function `step` as an interrupt service routine. For this example, the same technique will be used.

The first functions will be called `adc_filter` and its iputs/outputs will correspond with Fig. 1.3. In accordance with this Figure, the function definition will be as:

```
ADC_filtered = adc_filter(ADC)
```

This function will run at 700Hz, filtering data from ADC. The ADC signal will correspond to an actual position/revolution of potentiometer, attached to PicKit (described in Chapter 2). The filtered data will be one of several inputs to the second function called `step100` defined as:

```
[brake,direction,PWM] = step100(ADC_filtered,but1,but2,but3,but4)
```

This function will run at 100Hz and will transform the ADC signal to generate PWM, control the pushing of buttons and consequently manage the direction and break of the motor.

Another role of MATLAB will be to connect the generated files for both functions with already existing MPLAB project along with building and flashing of this project. All of this will be controlled from a simple graphical user interface (GUI).

1.2.3 Role of MPLAB

MPLAB will be used for creating the "base" for generated code. The most important points can be summarized as:

- All of the MCU configuration will be done there, which even for relatively easy task such as motor control is quite extensive.
- Three timers will be initialized. First for generating PWM, second for `adc_filter` interrupt and third for `step100` interrupt.
- Main function with endless loop containing interrupt-dependent calls of functions generated from MATLAB.

As the generated functions will be in separate files, the MPLAB will contain only the call to those functions. For instance, a code similar to one in Section 1.1.3 can be used:

```
int main(void) {
    configMCU();
    while(1){
            if (flag_100 == 1) {
                bool but1 = read_butt1();
                .
                .
                .
                bool but4 = read_butt4();
                step100(ADC, &brake, &dir, but1,...,but4, &PWM);
                flag_100 = 0;
            }
        }
    }
```

This code is only used as an example, however, the resulting code will use the same principle. In this exemplary code, the `flag_100` is a parameter, which is being set to `1` by the appropriate interrupt service routine. This way the function `step100` will be called with the desired timing. Also a combination with functions written in C is possible (e.g. `read_but()`). The function call for generated functions from MATLAB will be fixed and therefore has to contain the correct inputs with their order taken into account as well. This is because it has to fit with the generated function from MATLAB, where the fixed inputs have to be complied.

1.3 Testing functions for conversion

User's Guide for MATLAB Coder, among other things, states all the supported functions for conversion. Nevertheless, many of them have limitations for C/C++ code generation. [5]

Therefore, one of the last goals of this work will be to make a very simple testing of some of the supported functions and examine the generated code. Examples of analyzed topics are:

- Matrix operations
- Generating random numbers
- Trigonometric functions
- Fast Fourier Transform
- Finding Eigenvalues
- Singular value decomposition (SVD)
- Generating Objects

2 Hardware & Configuration

The goal of this chapter is to describe all of the used hardware (HW). The focus will be mostly on HW used for the double drive control as it is considered the main application in this thesis. Apart from HW, also the configuration of the microcontroller (MCU) will be described, as well as the context for double drive control. Providing such information should become a foundation, enabling an easy repetition of the experiment. The MCU configuration will be described in a way, which should allow for making simple changes to it by the potential user. Furthermore, it might as well be useful for people, who are not very experienced with the topic of microcontrollers. It should allow such people to move quickly through later chapters as well as to profit from other reference materials, like data sheets, Microchip's reference manuals, etc.

Figure 2.1: PIC EduKit used for the Double Drive control. [7]

2.1 PIC Edu Kit

The PIC Edu Kit is shown in Fig. 2.1. It consists of a set of electronic devices/boards designed to aid in classes of the department of Mechatronics at the Faculty of Mechanical Engineering, Brno University of Technology.

The Edu Kit offers a wide range of peripherals including digital inputs/outputs, analog inputs/outputs, communication protocols like SPI, I2C, RS232, USB to peripherals

enabling control of the Double Drive. The above mentioned makes the PIC Edu Kit a multipurpose tool for solving diverse amount of tasks. The main source for this section and the following text is [7], where more information about the PIC Edu Kit can be found.

For the purpose of this thesis, only several parts of the EduKit are important and will therefore be described. Those include components, which were involved with the Double Drive control, for example:

- 4 buttons
- LEDs
- dsPIC microcontroller
- Potentiometer

Some other parts/functionalities might be eventually used, but will not be described in separate section.

2.1.1 Buttons

In total, there are four pushbuttons on the PIC Edu Kit connected to digital I/O pins. All of them were used for the Double Drive to control the orientation of rotation and switching brake on and off. Use of each button is described in Table 2.1.

Table 2.1: Buttons on PIC Edukit and their use in Double Drive control.

Button Label	Pin Name	Polarity	Function
BUT1	RB4	LOW when pushed	sets brake on
BUT2	RA4		sets brake off
BUT3	RA9		changes direction to clockwise
BUT4	RC3		changes direction to counter-clockwise

The schematic for the connection of buttons is illustrated in Fig. 2.2. It can be seen that when the button is pressed, the I/O port is pulled down to ground (GND) and therefore a logic LOW (0) will be on the I/O port. When the pushbutton is released, there is a connection between its two legs, connecting the digital input pin to +3V3 via a 10kΩ resistor, so that a HIGH (1) can be read [7].

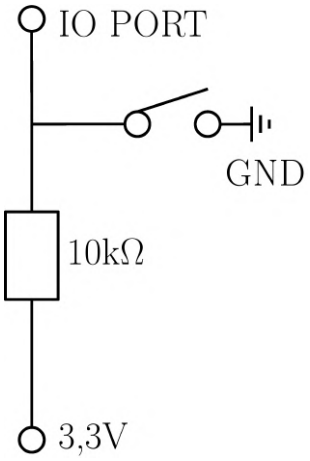

Figure 2.2: Schematic for the connection of buttons.

2.1.2 LEDs

The LEDs are used to signalize the status of Double Drive. In total, there are 3 LEDs for each motor to signal the status of brake (on/off), the direction of rotation and the PWM duty cycle. Since only one motor will be controlled, the brake for the second motor (indicated by LED3) will be permanently on. All the LED indications on PIC Edukit for Double Drive control are described in Table 2.2[1].

Table 2.2: LED indications on PIC Edukit for Double Drive control.

LED Label	Pin Name	Color	Function
LED4	RC7	Green	PWM duty cycle
LED5	RA10	Green	Indication of direction
LED6	RA7	Orange	Status of brake
LED3	RC8	Orange	Brake for the second motor

If the brake is on and the direction of rotation is clockwise, the relevant LEDs will be on. For the indication of duty cycle, the LED will get dimmer the lower the duty cycle is (for duty cycle equal to `0` the LED will be off)[2].

[1]The LEDs are actually on both the PIC EDU Kit and the Double Drive. This proved very convenient as the programs could be tested only with the use of PIC EDU Kit, without the Double Drive.

[2]The LED indicating PWM will never be completely off due to a limitation in the minimum value for PWM. Also in case of a high frequency PWM the LED might appear bright in the entire scale, so the indication has to be used with awareness.

2.1.3 Potentiometer

A single turn potentiometer PC1622NK010 (technical details can be found in [9]) is used. The schematic and principle is shown in Fig. 2.3. As illustrated in the scheme, this potentiometer is nothing but a resistor with one variable end. Detailed description of principle can be found in [10].

In the case of Double Drive control, the resistance/voltage drop between terminals two and three (see schematic in Fig. 2.3) is measured and transformed to PWM duty cycle. If the wiper is turned all the way to the left, the PWM duty cycle will be 100% and vice versa.

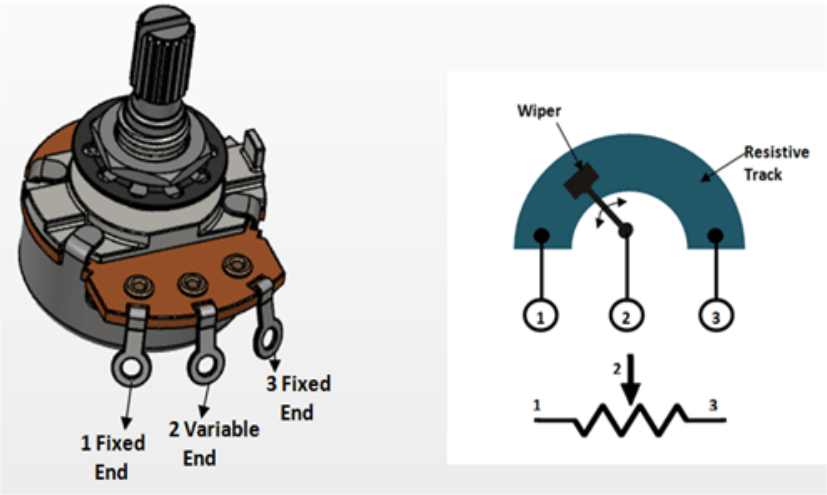

Figure 2.3: Schematic of potentiometer [11].

There are two analog outputs from the potentiometer. One of them is not filtered (connected to `AN4` pin) and the other is filtered by a simple RC filter (and connected to `AN5` pin) [7]. The non-filtered signal will be used since a simple filter is already implemented in the generated code from MATLAB.

2.1.4 dsPIC Microcontroller

The Digital Signal Controller incorporated in PIC EduKit is a dspic33FJ128MC804. It requires a supply voltage between 3,0 to 3,6V and features a maximum performance of up to 40 MIPS. Its peripherals include 10/12-bit analog-to-digital converter (ADC), many digital I/O pins, 2 UART modules for asynchronous serial communication, up to five 16--bit counters/timers (configurable to give two 32-bit timers and one 16-bit timer), up to four PWMs and more. [8] PIC microcontrollers, designed by Microchip Technology, are also considered a good choice for beginners [12].

A really useful feature of dsPIC is a peripheral pin select (PPS). Any pin labeled as RPx can be configured, e.g., for UART, SPI, PWM, timer input, interrupt, and more. Thanks to this feature all the functionality can fit into only 44 pins. This can be seen in Fig. 2.6, where the pin diagram is displayed. For example, pin 23 can be an analog input (AN4), a comparator input (C1IN-), remappable peripheral (RP2), a change notifcation input (CN6, which can generate an interrupt when an input changes state), or a digital input or output (RB2). [6, 8] Some of the pins were already described na Tables 2.1 and 2.2.

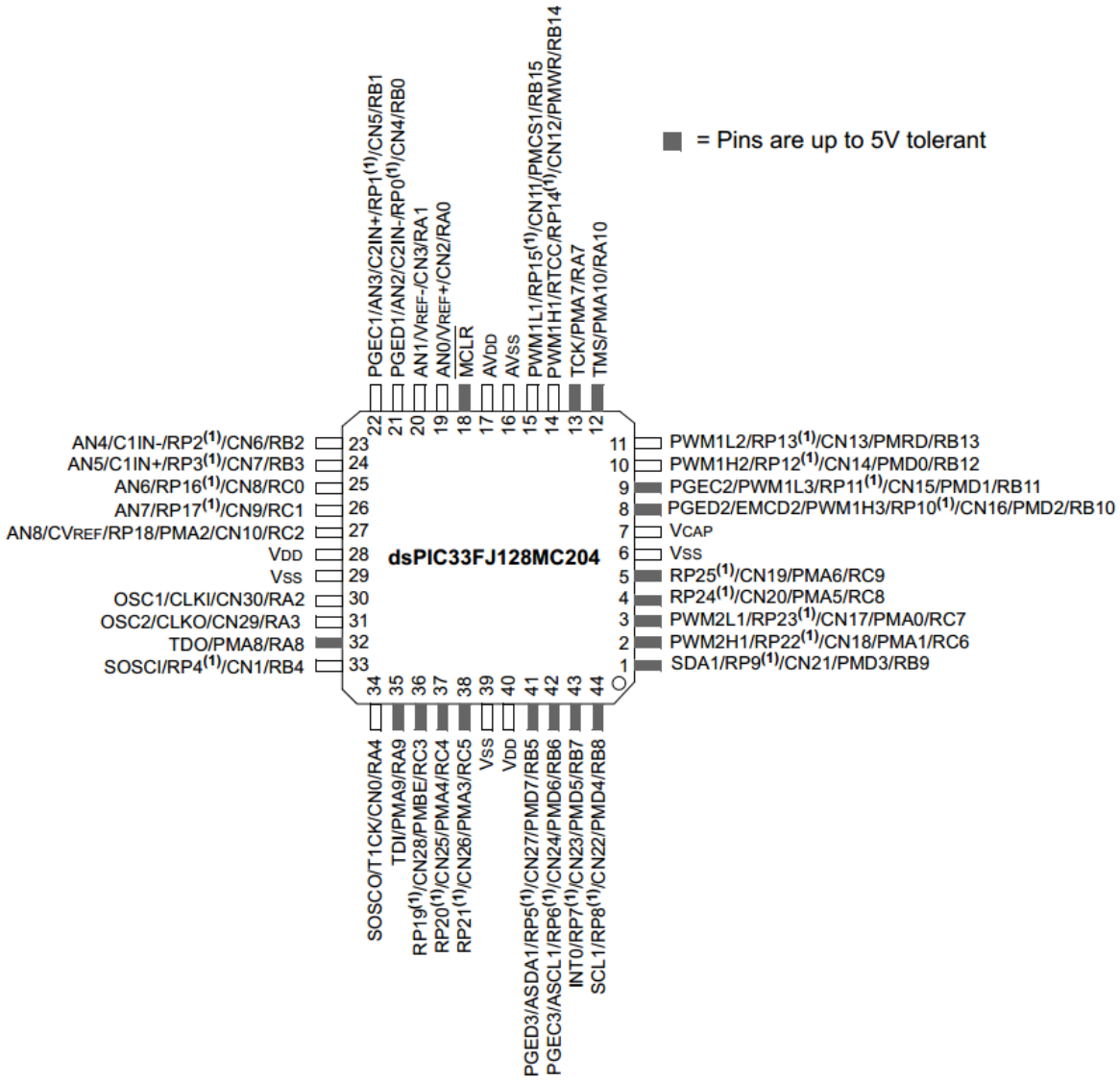

Figure 2.4: Pin diagram for dspic33FJ128MC804 microcontroller [8].

In the example of Double Drive control, the PWM for controlling the speed of the motor is generated by an Output Compare (OC) which is mapped to pin RC7 (pin 3). The OC uses Timer2 from MCU for comparing its value with the value of two OC registers (see [8]). The principle can be seen in Fig. 2.5. The OC1R (OCxR in the Fig. 2.5) is set

to 0. The state of the output pin changes when the Timer2 value matches the compare register value OC1RS. The compare register value OC1RS will be given by a function generated from MATLAB and used to modify the duty cycle of PWM. The OC Mode bits are set to 101 thus the OC module is in a Continuous Pulse mode. [5]

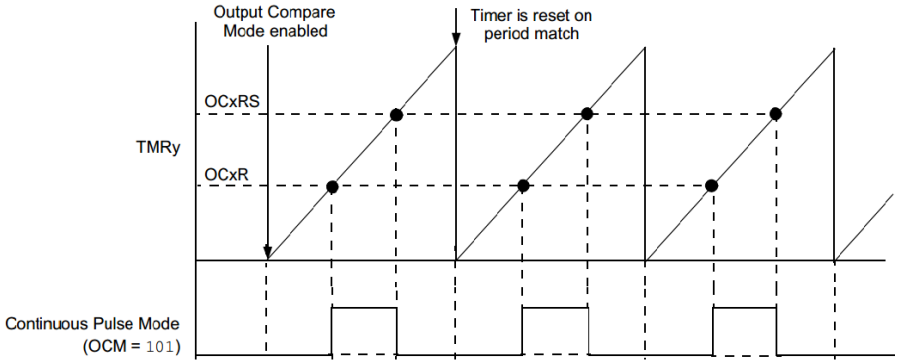

Figure 2.5: Principle of Output Compare for generating a PWM signal. [8]

2.2 PICkit 3

The PICkit 3 allows programming and debugging of dsPIC microcontroller[3] using the MPLAB X Integrated Development Environment (IDE). MPLAB IDE is a software program which runs on a PC to develop applications for Microchip MCUs. For further details about MPLAB IDE, see [14].

Figure 2.6: PICkit 3 Programmer/Debugger [15].

Further details about PICKit 3 Programmer/Debugger can be found in its User's Guide [16].

[3] Also many other PIC microcontrollers are supported, as stated in [13]

2.3 Arduino

Arduino UNO, which was used to analyze generated code from 'Simulink Support Package for Arduino Hardware' is shown in Fig. 2.7

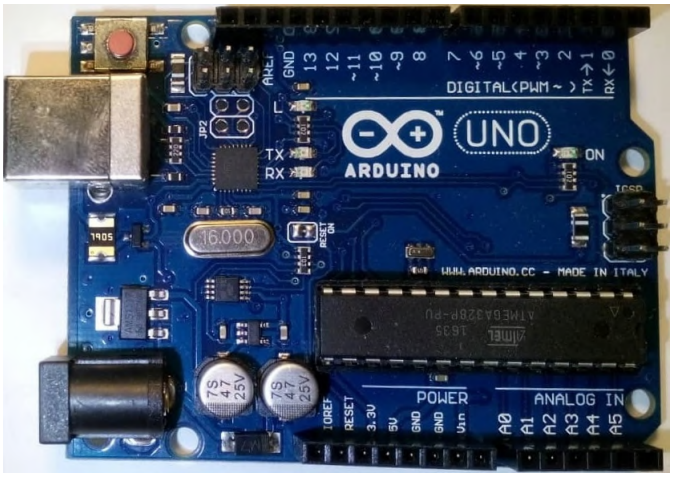

Figure 2.7: Arduino UNO used for code generation from Simulink.

It was connected with a type-B USB cable directly to the computer. To use Simulink for Arduino code generation, the target hardware board has to be selected in Tools » Run on Target Hardware » Options. The option is shown in Fig 2.8.

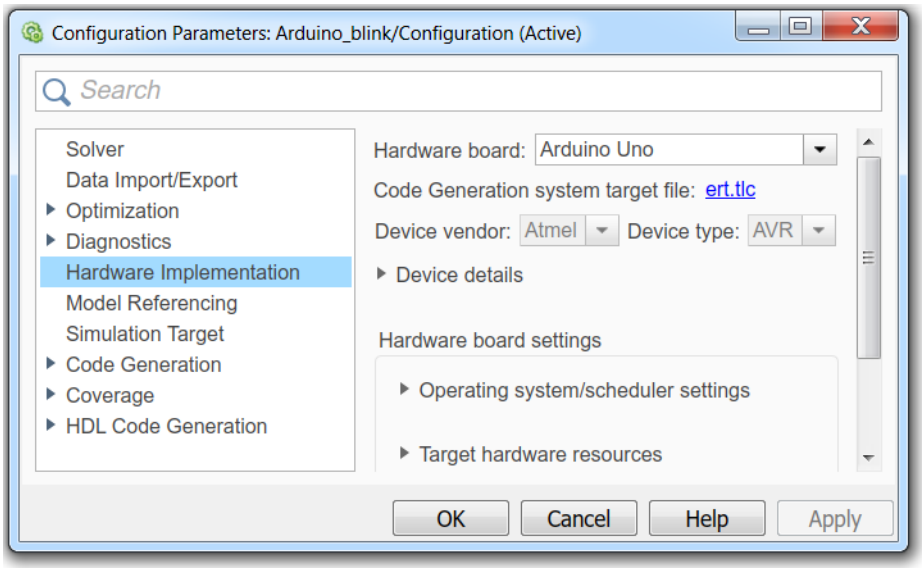

Figure 2.8: Configuration of Simulink for the use of Arduino UNO.

2.4 Double Drive

Double Drive (Fig. 2.9) is a laboratory model in MechLab (Mechatronics Laboratory) in Brno University of Technology. It consists of two brushed DC motors of PD3064 series with different planetary gearboxes [17]. Also several LEDs are used with the same functionality as was described in Section 2.1.2 for PIC EDU Kit. More information including basic motor properties can be found in [18]. The control circuit also consists of a LMD18201 H-Bridge, which will be discussed in the next section.

Figure 2.9: Double Drive - a laboratory model consisting of two brushed DC motors.

2.4.1 H-Bridge

H-Bridge is a circuit that can be used to control the speed and direction of a brushed DC motor. The LMD18201 H-Bridge, used in the Double Drive, is shown in Fig. 2.10.

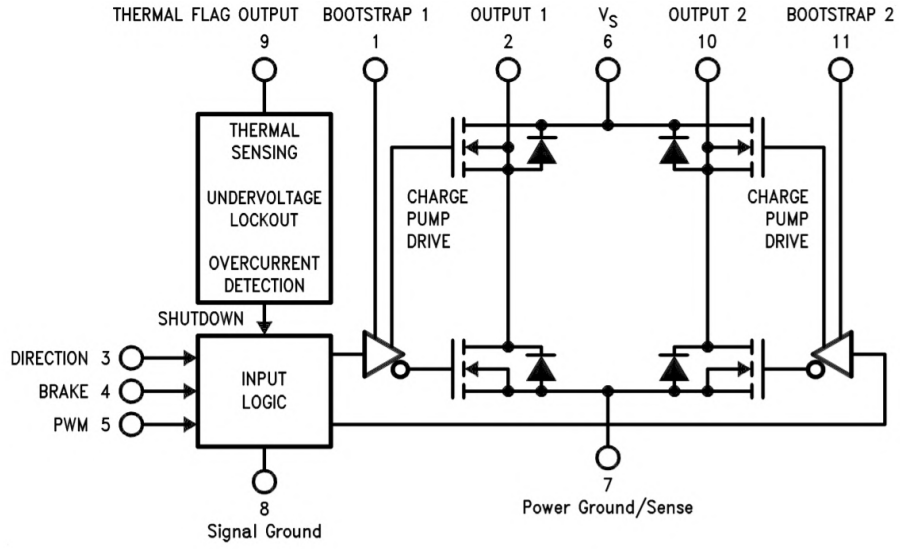

Figure 2.10: Functional Block Diagram of LMD18201 H-Bridge [19]

Full pin description can be found in [19]. However, there are 3 pins worth noting as they will be later used as outputs from MATLAB functions generated for Double Drive control. Those pins are:

- **Pin 3, DIRECTION**: the input in this pin controls the direction of rotation of the motor load.
- **Pin 4, BRAKE**: this input is taken to a logic high if braking is desired. It is also necessary to apply logic high to PWM input (pin 5) in this case. The DIRECTION input (Pin 3) also plays role in the BRAKE functionality. If Pin 3 is high, both current sourcing output transistors are ON and if Pin 3 is low, both current sinking output transistors are ON. To turn off all output transistors a high is applied to BRAKE and PWM input (pin 5) is set to low. In this case only a small bias current of approximately -1.5 mA exists at each output pin. [19]
- **Pin 5, PWM**: this input depends on the type of PWM signal. [19] presents 2 types, from which a sign/magnitude PWM is used and will later be described.

Sign/magnitude PWM is demonstrated in Fig. 2.11. It consists of a separate direction and an amplitude signals. The magnitude signal depends on the duty-cycle, and a continuous logic low level or the absence of a pulse signal results in zero drive. Current through the load (motor) is then proportional to the pulse width. The working principle of H-Bridge itself is very well explained in [20] and thus will not be further discussed here.

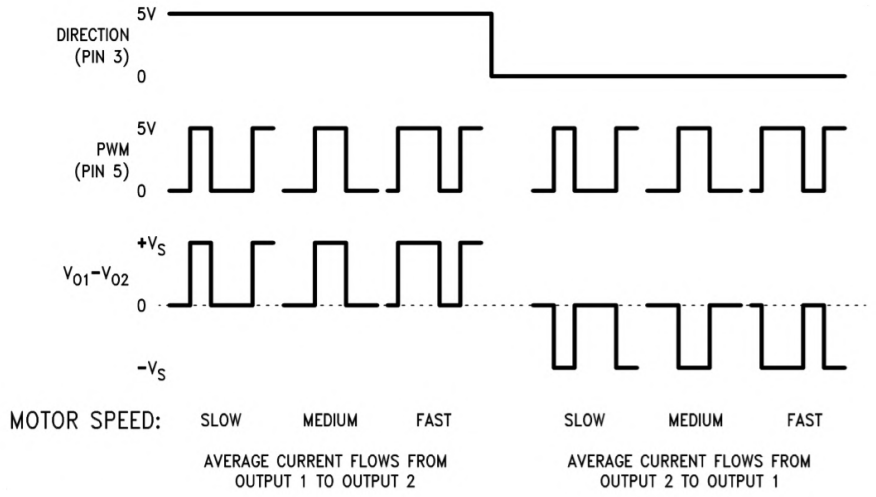

Figure 2.11: Sign/Magnitude PWM Control of LMD18201 H-Bridge [19]

3 Introducing MATLAB Coder

As has already been mentioned, MATLAB Coder generates C and C++ code from MATLAB code. After some actions taken beforehand, most of the MATLAB language and, despite some limitations, more than 45 toolboxes are supported. The generated code is readable and portable and, as illustrated in Fig. 3.1, can be integrated into projects as a source code, static libraries, or dynamic libraries. The generated code can be also packaged as a MATLAB executable (MEX) file, which can be called directly from MATLAB and used in the MATLAB environment either for verification or acceleration. [21, 22]

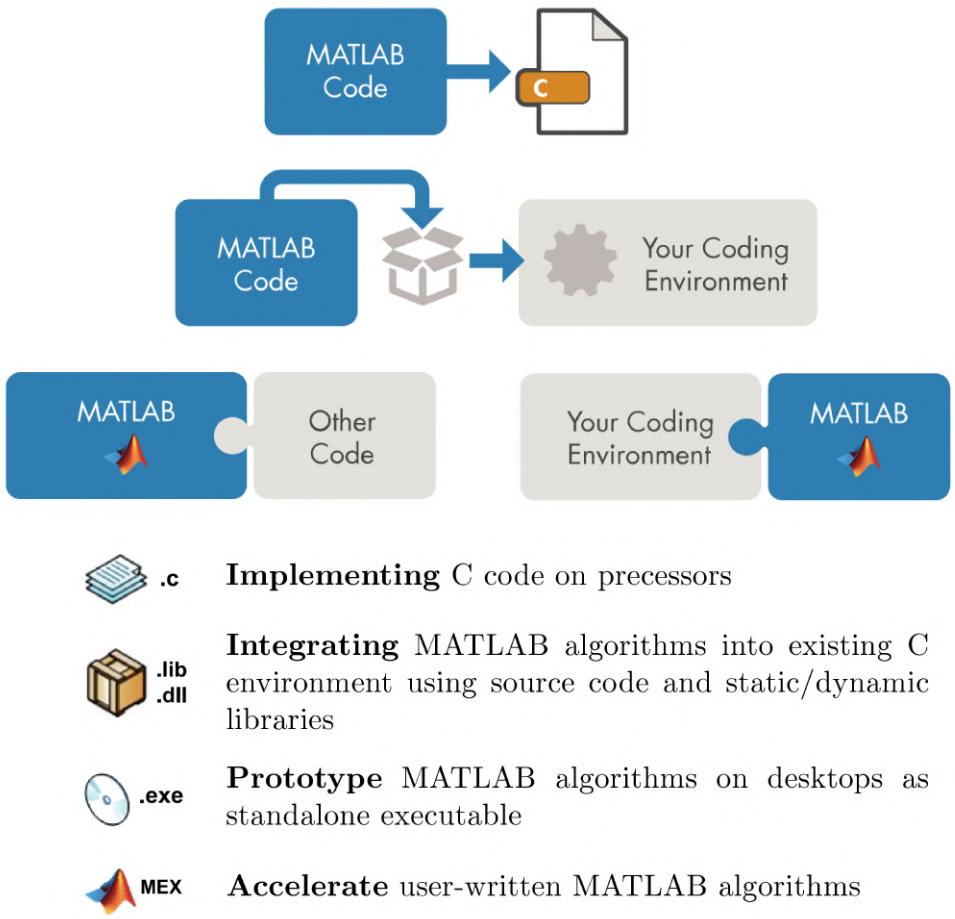

Figure 3.1: Different ways of utilizing MATLAB in code generation. Modification of schemes in [23] and [24].

In this chapter a short overview of MATLAB Coder will be presented, starting by a brief history of C code generation development, up to the more practical part, where some useful functions will be described. Nevertheless, as some of the topics are better explained on an example, this section can be considered as only an introduction for the next chapter, where MATLAB Coder will be put into practice with additional comments and tips.

3.1 Brief History

> *The first MATLAB was not a programming language; it was a simple interactive matrix calculator. There were no programs, no toolboxes, no graphics. And no ODEs or FFTs.*
>
> Cleve Moler

The origin of MATLAB goes back to years between 1965 and 1970 [25], and the very first version was made by Cleve Moler[1]

Likewise, C code generation from MATLAB is no novelty. Arvind Ananthan, a product marketing manager for MATLAB Coder and Fixed-Point Toolbox, describes the "MATLAB to C"development on a MathWorks blog [27] in accordance with following text:

- **2004** - The Embedded MATLAB Function block in Simulink was introduced. It allowed to incorporate MATLAB m-code into a Simulink model. Since version R2011a it was renamed to only MATLAB Function Block, but the functionality did not change. [28]

Figure 3.2: MATLAB Function block in Simulink.

- **2007** - The emlc function in Simulink Coder (formerly known as Real-Time Workshop) to generate stand alone C code from MATLAB was introduced.
- **2011** - Release of MATLAB Coder, the first stand-alone product from MathWorks to generate portable and readable C and C++ code from MATLAB code.

Currently, also an extension to MATLAB Coder and Simulink Coder exists. It is called Embedded Coder and it enhances MATLAB Coder with advanced optimizations for precise control of the generated functions, files, and data. [29] The use of MATLAB Coder with Embedded Coder can also improve code efficiency and customize the generated code [5]. A short overview of all the mentioned "coders"(Simulink, MATLAB, Embedded), can be found in [30].

[1] Cleve Moler is chief mathematician, chairman, and cofounder of MathWorks [26].

3.2 Useful Functions

This section describes the functions, which are necessary or which were later found helpful in generating C code from MATLAB. Some information about functions listed below can be found in MATLAB Coder User's Guide [5], however, most of the text in this section is based on the practical experience and application. The functions found as important are documented in the Table 3.1.

Table 3.1: Useful functions for C code generation from MATLAB, particularly when using the command line option and not the MATLAB Coder App.

Function	Description
`coder.ceval`	Calls an external C function, either from a separate C code or eventually from the previously generated C code. For example: `ADC_val = coder.ceval('read_ADC');` could use C function "double read_ADC()"
`coder.cinclude`	Includes a header file in the generated code. This is useful in combination with `coder.ceval`, if the C code requires also a header file.

`coder.target`	`target` variable is empty when running the MATLAB interpreter. This allows for defining different processing in the MATLAB function e.g. separating code only for MATLAB and code for code generation. Example:
	`if isempty(coder.target)`
	`plot(x,y)`
	`% This happens in MATLAB`
	`else`
	`% This happens in code generation`
	`end}`

`coder.Constant`	Defines value/s that are constant during code generation.
`coder.config`	Creates a configuration object in which all of the options for generating C code can be set.
`coder.typeof`	Defines the argument types for function which is to be converted to C code. If the function has more than one input, the argument types can be placed in a cell.
`codegen`	Invokes MATLAB Coder for generating C code. Usually combined with the `coder.config` and `coder.typeof`.

An example of the last 3 commands is presented in section 4.2, where they are used for generating C code in the Double Drive control.

3.3 Order of Evaluation in Expressions

The order of evaluation within an expression is not respected in the generated code from MATLAB Coder. Even though this poses no significant influence for the use in this thesis, it was found important to mention it. The effect could become an issue for expressions with side effects as the generated code may produce the side effects in different order from the original MATLAB code. Some examples, as stated in [31], that produce side effects include:

- Modifying persistent or global variables
- Displaying data to the screen
- Writing data to files

To avoid possible errors and to obtain more predictable results, there is a good coding practice to split expressions which depend on the order of evaluation into multiple statements. So for example an expression:

```
1 x = f1() + f2();
```

can be rewritten as:

```
1 x = f1();
2 x = x + f2();
```

which will ensure, that the generated code will call `f1()` before `f2()`.

3.4 Data Types

In MATLAB, all the variables can change their data types dynamically at run time[2], so the same variable can be used to hold a value of any class, size, or complexity [32]. For example, the following code works in MATLAB:

```
1 if condition == 1
2 some_output = ADC;  % ADC is type double
3 else
4 some_output = brake;  % brake is type logical
5 end
```

[2]Nevertheless, for many years, MATLAB had only one numeric data type with IEEE standard 754 double-precision floating point, stored in a 64-bit format. As the use of MATLAB extended its use to further applications and larger data sets, more ways to represent data were provided. [25]

However, in MATLAB Coder App the output types are derived from the inputs. Code generation does not support changing data types through assignment[3]. This means that it is not possible to use this code for code generation as it assigns a 'double' type of `ADC` to `some_output` if the condition is met. On the contrary, it uses a 'logical' value of `brake` if the condition is not met. Since such a change might be desirable, the script can be fixed by changing the data type of brake and hence writing the code as:

```
1 if condition == 1
2 some_output = ADC;
3 else
4 some_output = double(brake);
5 end
```

which is suitable for code generation. MATLAB Coder can use built-in C data types or predefined types from `rtwtypes.h` which can be found among the generated files after code generation. The data types can be explicitly specified in the project settings dialog box and also by the command line as will be shown in Chapter 4. Supported data types, which can be used, as shown in the example above, are listed in Table **??**.

Table 3.2: Supported data types. [33]

Data Type	Description
`double`	Double precision floating point
`int8, int16, int32, int64`	Signed integer
`uint8, uint16, uint32, uint64`	Unsigned integer
`logical`	Boolean true or false
`char`	Character array
`complex`	Data with real and imaginary components
`single`	Single-precision floating point
`struct`	Structure

In the next Chapter in Fig. 4.2 an option for fixed-point conversion can be selected. Although it was not used for the purposes of this thesis, it might still be useful. For more information about converting floating-point to fixed-point see [34, 35].

3.5 Calling by Reference

Passing by reference is an important technique for C/C++ code integration. When data is passed by reference, the program does not need to copy data from one function to another but uses addresses instead. [36] It is not possible to return several outputs from a function

[3]or any other statically-typed language like C, which must be able to determine variable properties at compile time.

in C language in the same way as in MATLAB. In case there are several outputs from a MATLAB function, which is to be converted to C, there are two possible outcomes. Both of them were tested and will be demonstrated below.

First, the function can return value of one of the outputs and then handle the rest of outputs as pointers. An example of a MATLAB function with this effect can look like:

```
function [brake,dir,PWM] = step100(ADC,brake,direction)
```

which after the code generation results in the following function definition:

```
double step100(double ADC, bool *brake, bool *dir)
```

The function now returns the value of `PWM` and handles `brake` and `dir` by reference (pointers). This happens every time, when the function has:

- Only one output.
- Several outputs, from which all but one are used as inputs and outputs simultaneously.

To force call-by-reference for all the outputs, several steps can be taken:

- All of the outputs are also declared as inputs.
- Two or more outputs are not used as inputs.

Since those two conditions might not be easily achieved for some functions, another output can be added to fulfill the second condition. This simple modification will ensure, that all of the outputs in generated code will be called by reference. The previously used function could be modified in the following way:

```
function [brake,dir,PWM,status] = step100(ADC,brake,direction)
```

which will result in function returning `void` and handling outputs by pointers:

```
void step100(double ADC, bool *brake, bool *dir, double *PWM, double ...
    *status)
```

Here, the variable `status` is simply returning value of `1` and the rest of the function remains the same, which will not reduce the performance. Also note, that in C all arrays are always passed by reference, so there is no need for using this technique on such arguments.

4 Integrating MATLAB Coder into Double drive control

There are two ways how MATLAB Coder can be utilized in code generation. Either the MATLAB Coder App can be used, or everything can be programed by commands in a script.

In the first option, MATLAB Coder App guides the potential user through the process of code generation. This is probably the most straightforward way to understand code generation from MATLAB, especially for people new to this topic.

The second option offers much faster testing and responses to any changes in code for conversion to C since most of the script remains the same. By changing only a small portion of the script, the code with, e.g. settings, can also be used for several individual tasks.

In this chapter, both of these methods will be described and applied on a specific example of a double drive control to demonstrate their use.

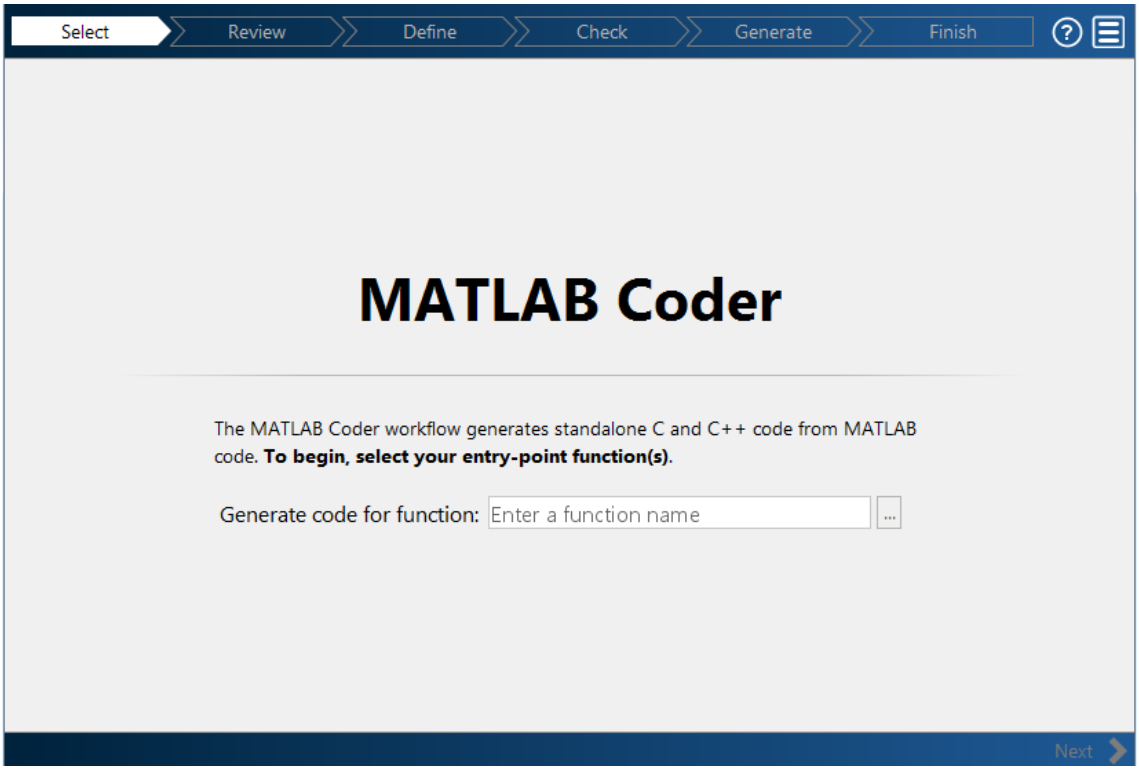

Figure 4.1: Initial page of MATLAB Coder App.

4.1 MATLAB Coder App

To use MATLAB Coder App manually, the following procedure can be used. First of all the MATLAB code, from which the C code is to be generated, has to be provided[1]. This code has to be written as a function (scripts can not be converted). After starting the MATLAB Coder App, a window (Fig. 4.1) will be displayed. There the function for code generation has to be selected.

4.1.1 Setup of numeric conversion

After selecting the function a project file "xxx.prj" will be created in your current folder, where "xxx" stands for the name of the function. It contains information and settings of created project. In this case function "step100" was used (as can be seen in Fig. 4.2).

In case that the ".prj" file already exists, e.g. the app is used for the same function more than once, a warning will be displayed that the specified project already exists. Then it is either possible to reopen the existing one, enter a different name, or overwrite it.

It is also possible to set up the numeric conversion and 3 alternatives are offered (see Fig. 4.2). Default option is "None" which was also selected in this case. That way the generated code will use predefined data types from the next step.

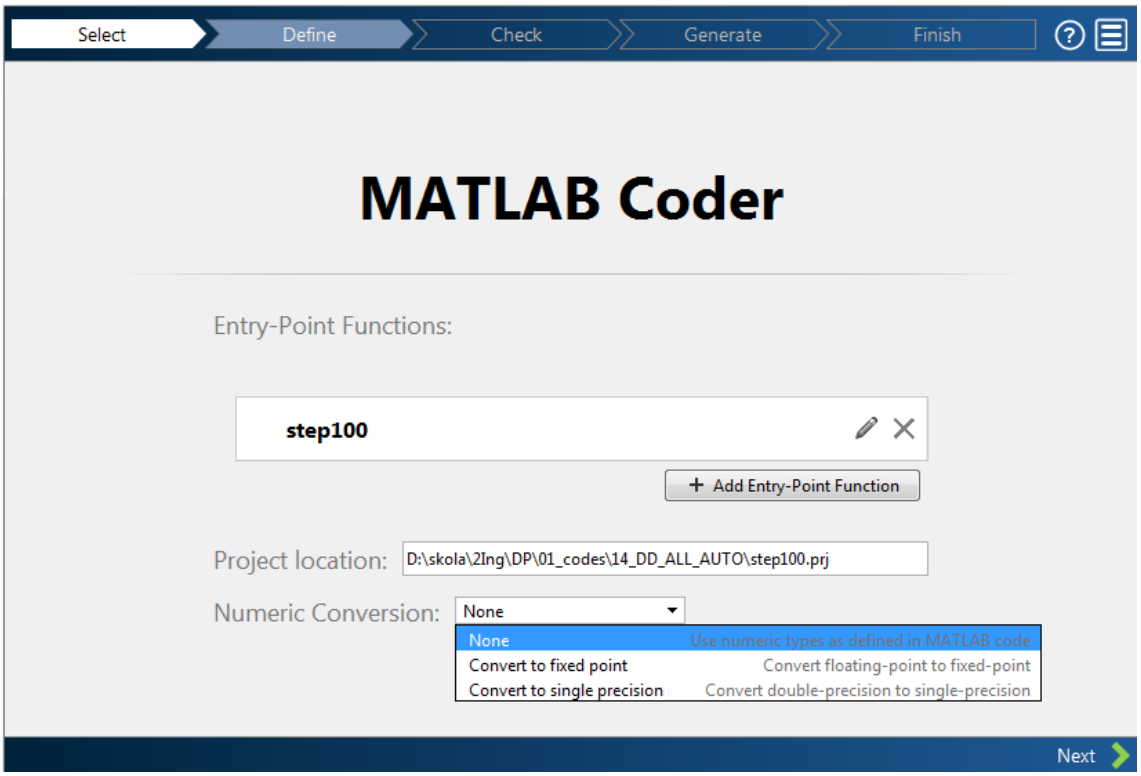

Figure 4.2: Setting up numeric conversion.

[1]Note that the code can not use any built-in functions or toolbox functions which are not supported for code generation. To find out which functions are supported see [37] for alphabetical list or [38] for category list of supported functions.

Note that single precision conversion can not be effectively used for all MATLAB functions without further changes, i.e., as such they can still be used, however, they will use double precision. One of such functions is for example `abs(x)`, warning generated for this function is shown in Fig. 4.3. The fixed-point conversion was already commented in previous Chapter, and the process of using this option in MATLAB Coder App can be found in [39].

1	⚠	step100	6	The function abs uses double-precision in the C89/C90 (ANSI) standard. For single-precision code, consider using the C99 (ISO) standard or use your own function.

Figure 4.3: Warning for `abs(x)` when a single precision conversion is selected.

4.1.2 Defining input types

In the next step, the input and global types have to be defined (Fig. 4.4). Input and global types can be entered directly, however, it is very convenient to use a test file for this cause. Test file, sometimes also called as testbench, is a file (MATLAB script) that uses function for conversion with specified input types. An example of such a script may look like this:

```
1 [brake,direction,PWM,status] = ...
      step100(double(1),true,false,true,false,true,false)
```

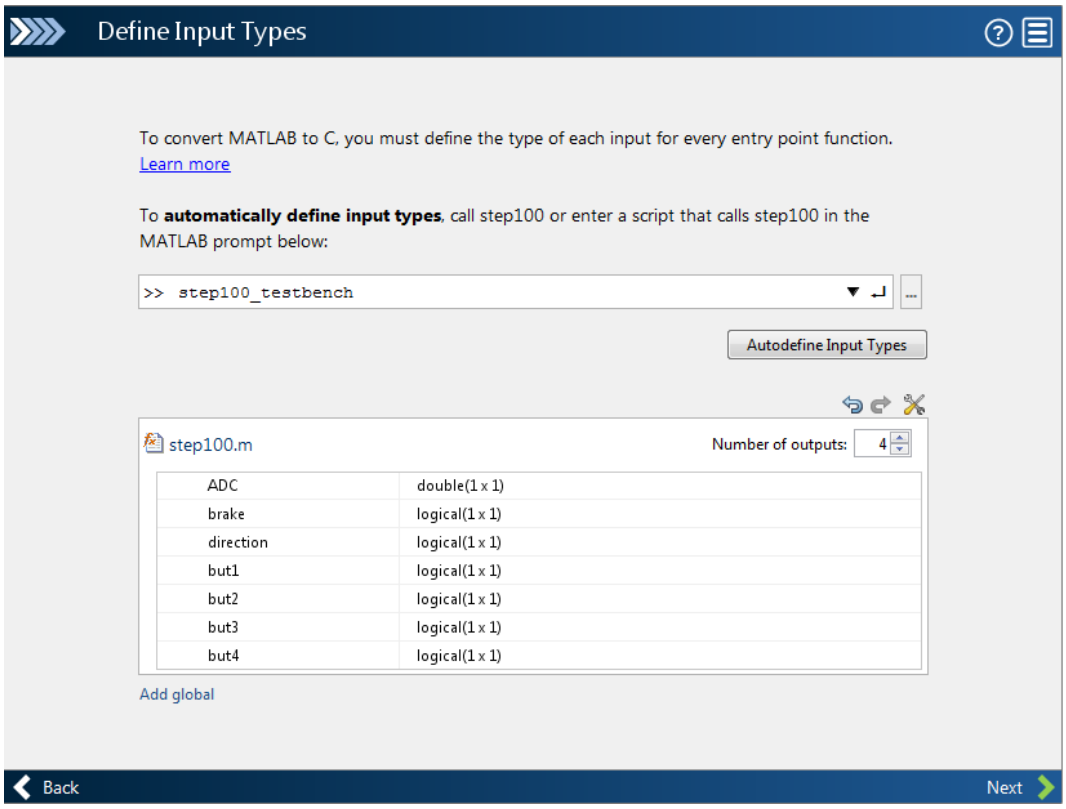

Figure 4.4: Defining input types.

In the MATLAB Coder app, the test file is then selected and an option to "Autodefine input Types" is used. This way all the input and global types are entered at once (Fig. 4.4) and any later changes to the code can be easily applied. Supported data types are listed in Table **??**.

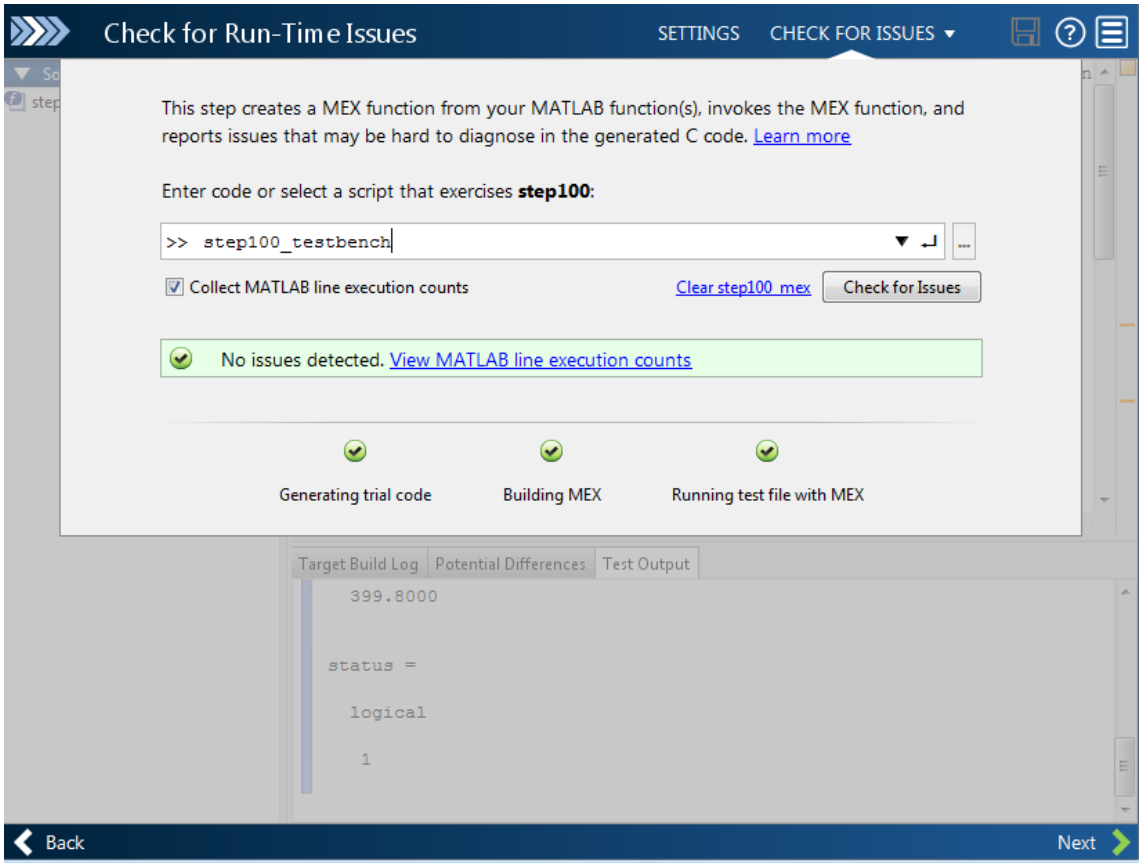

Figure 4.5: Checking for run-time issues.

4.1.3 Checking for run-time issues

Fig. 4.5 shows another step in MATLAB Coder App, which identifies unsupported functions or syntax and also checks for any issues including the one specified in section 3.4.

To perform the check the already created testbench can be used to exercise function for conversion or a function call with defined input types can be provided, such as:

```
1       step100(double(1),true,false,true,false,true,false)
```

MATLAB Coder then builds a MEX file providing an interface between MATLAB and function in C. The MEX file is then tested and if any issues are found, it provides warning or error message. The problematic code will eventually be highlighted in a window in which the code can then be edited. In Fig. 4.5, the App does not detect any issues.

It is also possible to skip this part by directly clicking "Next", e.g. in cases, where this check has already been done followed by a small change of code etc. However, it is a best practice to perform this step [5].

4.1.4 Settings and code generation

Last step before code generation is the setting and choosing the build type and language (C or C++). It is recommended to uncheck the option for supporting non-finite numbers (More Settings » Speed » Suport non-finite Numbers) unless it is required by the application. This will reduce the number of generated files as well as improve code readability. The rest of the setting can be left at default (some other options will be discussed in Section 4.2).

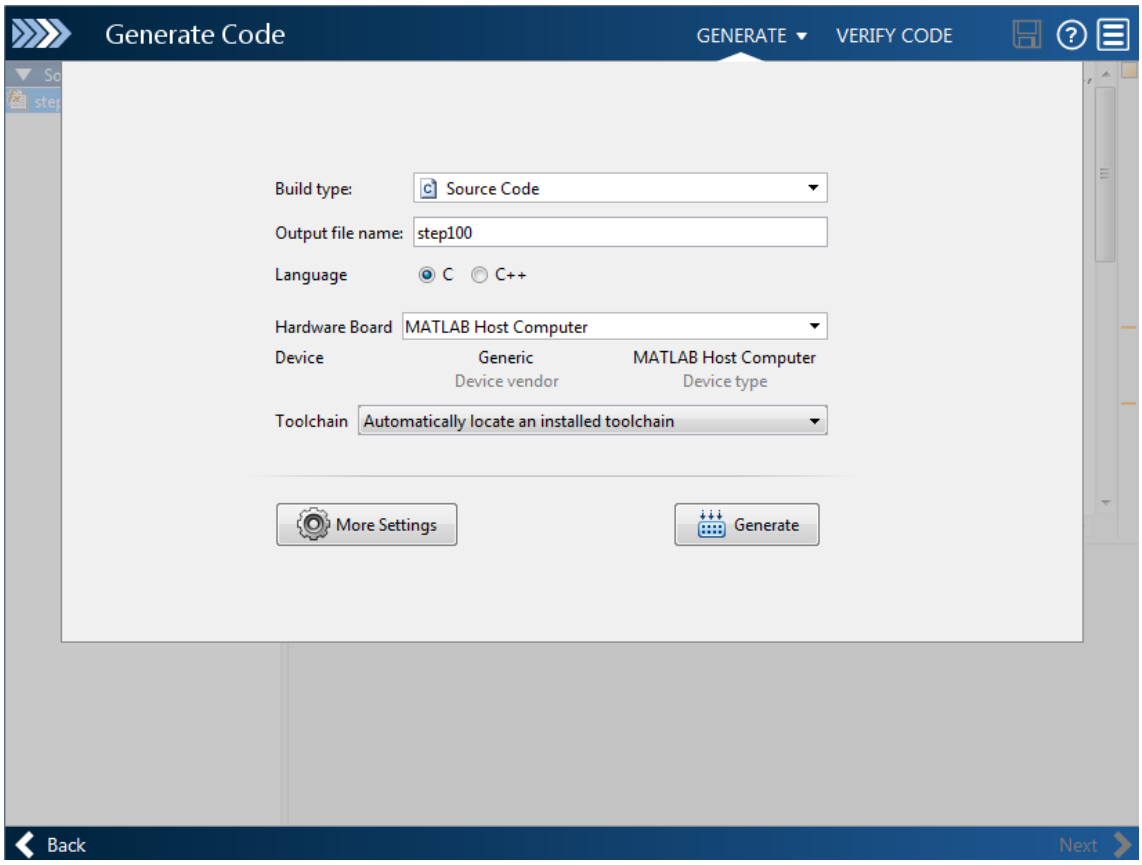

Figure 4.6: Settings and generating code.

4.1.5 Reviewing generated code

Fig. 4.7 shows the review of successfully generated code. This report can be also found as .html file in "project_name » codegen » lib » step100 » html » index.html". Here the generated files can be inspected. Finally, in Fig. 4.8 the project summary can be seen. In this step, it is also possible to package the project, i.e., the generated files to a `.zip` file.

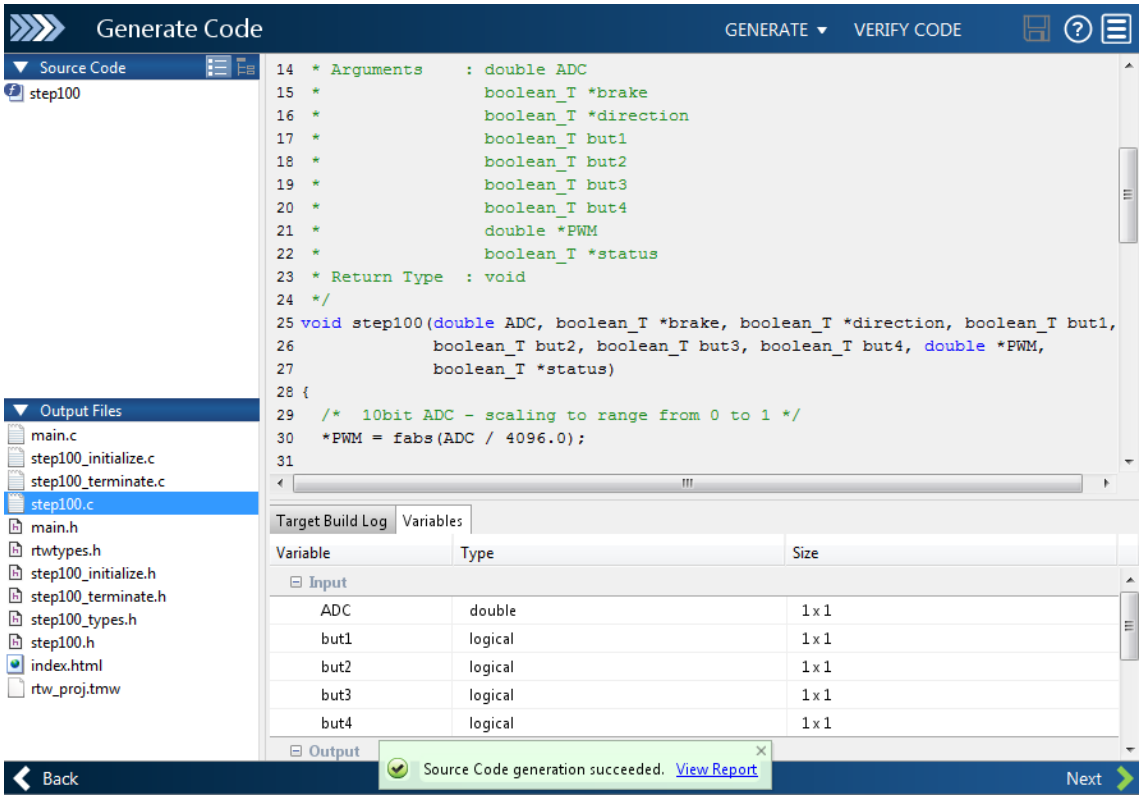

Figure 4.7: Successful code generation.

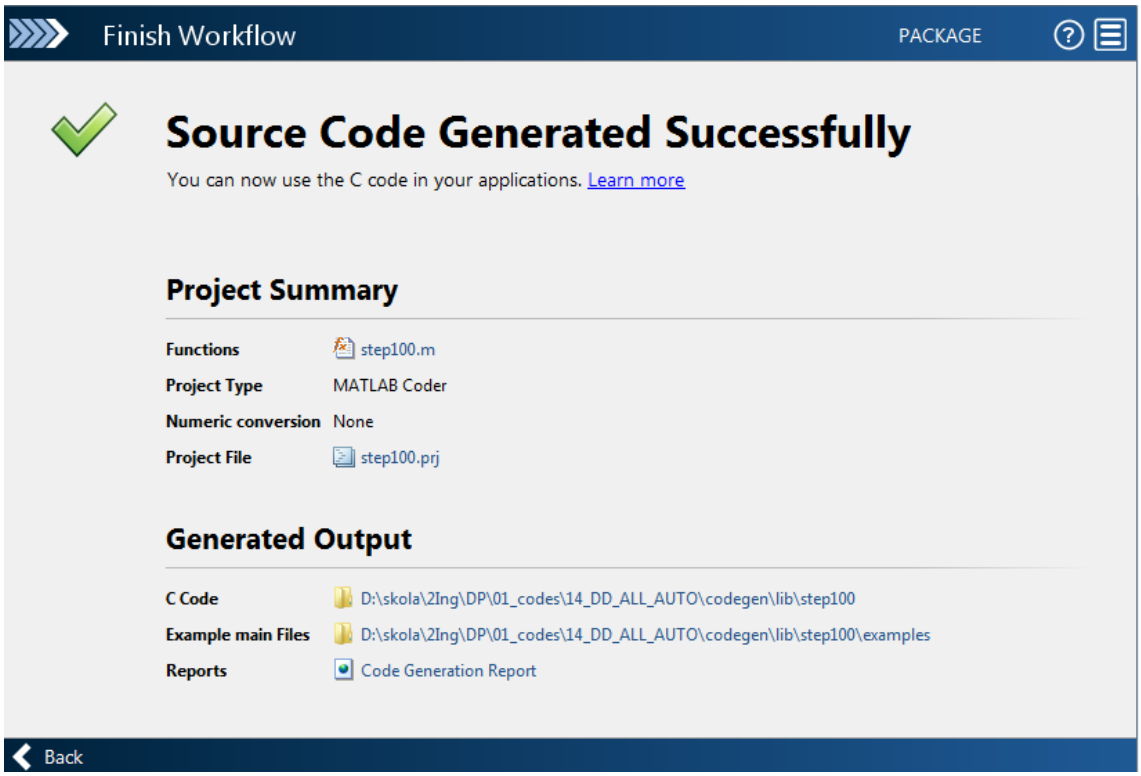

Figure 4.8: Final workflow.

4.2 Code generation using command line

In the previous section a way to use MATLAB Coder App was shown. Despite it is relatively easy to use and gives immediate feedback, it is quite time consuming to go through all the steps manually. This applies especially to cases, where more than one function is converted to C and all the more if there are frequent changes to the original code. Using command line circumvents the time consumption problem and offers easy integration to existing MATLAB code.

This method was preferable also in the case of Double drive control. Not only more than one function was generated at the same time (as shown in Fig. 1.3 in section 1.2), but the code generation was also combined with another MATLAB code, which dealt with automatic building and flashing (further discussed in section 4.3).

To avoid using MATLAB Coder App, there were several steps which had to be made. Basically everything which was set up in the App has a command line equivalent. Necessary steps for command line use will be described in the following sections with application on the Double Drive code generation.

4.2.1 Creating configuration object

First a configuration object has to be created. In a general case it can be called as `cfg`. There should be a unique configuration object for each function which will be converted to C. However, if the setting is the same for both or several functions then it is possible to use only one object. Below is an example of configuration object for function `step100` which was used for Double drive control:

```
%% Creating configuration object for function step100
step100_cfg = coder.config('lib');
step100_cfg.GenCodeOnly = true;
step100_cfg.GenerateCodeMetricsReport = true;
step100_cfg.GenerateReport = true;
step100_cfg.ReportPotentialDifferences = false;
step100_cfg.MATLABSourceComments = true;
step100_cfg.SupportNonFinite = false;
```

Note that all of the parameters except `coder.config('some_parameter')` are optional and that code generation will work even without specifying them. In that case a default setting will be used. However, their use can affect the resulting code in many beneficial ways, as will be described below.

```
cfg = coder.config('lib');
```

This line is the most important and cannot be omitted. It tells MATLAB Coder whether to generate MEX function, static library (`.lib`), dynamic library (.dll) or executable (.exe). In this case a static library was chosen as it also generates desired C code.

```
cfg.GenCodeOnly = true;
```

As only the C files are needed for the needs of this thesis, the creating of static library can be prevented from generation by this parameter. If set to true, only individual C files will be generated. If set to false or not specified at all, a `.lib` file will be generated along with the C files.

```
cfg.GenerateReport = true;
```

Setting the `GenerateReport` to `true` allows for an easy and quick review of the generated code. Checking the resulting code is always recommended.

```
cfg.MATLABSourceComments = true;
```

This option was found very useful as it inserts MATLAB source code as comments into the generated C code. This allows easy verification and helps with orientation in the generated code.

```
cfg.SupportNonFinite = false;
```

This option has already been mentioned in section 4.1.4. Since there was no need for supporting non-finite numbers, this parameter was set to `false`, thus reducing the number of generated files.

4.2.2 Defining argument types

In the next step, the data types have to be defined. Desired data types are written in a cell array and then used in the `codegen` command (discussed in the next section). The cell array is defined as `cell(n_in,1)`, where `n_in` stands for the number of inputs (empty cell array can be used, if the function takes no input arguments). The order of elements is also important and it has to correspond with the order, in which inputs appear in the function for conversion. Definition of `step100` is as follows:

```
1 function [brake,direction,PWM,status] = ...
      step100(ADC,brake,direction,but1,but2,but3,but4)
```

There are 7 inputs to this function and all of them, except `ADC`, are of type logical, therefore defined as boolean `true`/`false`. `ADC` is of type `double` which is a default data type and can be typecasted by value `0`. Defining the argument types using the command line then looks like:

```
1  %% Defining argument types for function 'step100'.
2  step100_ARGS = cell(1,1);
3  step100_ARGS{1} = cell(7,1);
4  step100_ARGS{1}{1} = coder.typeof(0);
5  step100_ARGS{1}{2} = coder.typeof(false);
6  step100_ARGS{1}{3} = coder.typeof(true);
7  step100_ARGS{1}{4} = coder.typeof(false);
8  step100_ARGS{1}{5} = coder.typeof(true);
9  step100_ARGS{1}{6} = coder.typeof(false);
10 step100_ARGS{1}{7} = coder.typeof(true);
```

Second function which is to be converted for this example is `filter` with the following definition:

```
function [filtered] = adc_filter(noisy)
```

This time, there is only one input parameter, which is once again of type `double`. The interpretation will thus be analogous:

```
%% Defining argument types for filter
adc_filter_ARGS = cell(1,1);
adc_filter_ARGS{1} = cell(1,1);
adc_filter_ARGS{1}{1} = coder.typeof(0);
```

In addition to the previous plan, a third function will be generated. It will be used for setting up the PWM frequency, therefore some part of the MCU configuration will be possible to change from MATLAB as well. This step also shows different way of defining data types as a constant value will be used for code generation. The function definition is:

```
function [period] = pwm_PR()
```

This function is simply returning value for a timer controlling the PWM frequency. Its value is set before the code generation takes place and will remain constant afterwards. The value of `period` will be set in GUI created for Double Drive control (discussed later) and the script for code generation will use that value. The commands for defining the global `PR` was designed as:

```
%% Defining global type of PR for function pwm_PR.
GLOBALS = cell(1,1);
GLOBALS{1,1} = coder.Constant(uint16(PR));
```

Also note that the name of the cell (in this case `GLOBALS`) containing the argument types is optional.

4.2.3 Invoking MATLAB Coder

Once the initial setting and argument types are put in place, the MATLAB Coder can be invoked. This is done by a `codegen` command. The first argument is `-config ... config_object_name`, second is the function name, third the `-args args_cell_name` and last `nargout` stating the number of outputs. In accordance with previous section, the MATLAB Coder command line option for function `step100` can be used as:

```
%% Invoke MATLAB Coder for step100 function.
codegen -config step100_cfg step100 -args step100_ARGS{1} -nargout 4
```

likewise for function `filter`:

```
%% Invoke MATLAB Coder for filter function.
codegen -config adc_filter_cfg adc_filter -args adc_filter_ARGS{1} ...
    -nargout 1
```

and also for function `pwm_PR` which uses the global variable `PR`:

```
%% Invoke MATLAB Coder for pwm_PR function.
codegen -config cfg -globals {'PR',GLOBALS{1,1}} pwm_PR
```

By combining all the codes described in this section, the same effect as by using the MATLAB Coder App graphical interface can be obtained. However, in this case 3 functions will get converted to C at the same time, which proves very advantageous. Another benefit lies in the possibility of using the above mention code in different MATLAB functionalities. For example some personalized GUI can be made where the settings can easily be changed. Eventually this code can also be combined with commands for automatic building and flashing MPLAB project to a MCU as will be described in the next section.

4.3 Building and flashing MPLAB project from MATLAB

There are various ways to build and flash MPLAB project from MATLAB. This topic was also discussed with Lubin Kerhuel[2], who has immense knowledge in this field and also with my colleague and friend Štěpán Zouhar, who eventually discovered a very simple approach. This technique was modified to fit the needs of this thesis and individual steps of the process will be discussed in this section.

In section 4.2 the possibility to generate a C code by only using MATLAB script was described. The same applies for copying generated files to existing MPLAB project as well as for its building and flashing to the MCU[3]. Even though that all of this can be made in MATLAB, the first step, which includes adding paths[4] to an environment variable is recommended to perform manually. This is because it is not fully supported to change it using a command line. Although there is a command to do so, it only supports 1024 characters, which might not be enough in some cases. Such a command would look like:

```
% WARNING - only use this if 'path' does not contain more than 1024 ...
    characters including the added_path! Otherwise set the ...
    environment variable manually
[status,cmdout] = system('setx path "%path%;added_path" /M')
% %path% represents the original path which is to be merged with ...
    'added_path'
% /M sets the path to system environment variable level instead of ...
    user level - This command has to be run as administrator
```

[2]His website can be visited through the link included in [3]

[3]To build and flash MPLAB project from MATLAB a command "system" is used, sending its parameters to a command line. More information can be found in [41].

[4]MPLAB X IDE automatically adds the directory containing the `make` (discussed in the next section) to its own path variable. However, to build outside of the IDE, the directory to the PATH environmental variable has to be added. [43]

where `added_path` is one string containing all of the following paths (here, for the sake of clarity written separately).

```
'C:\Program Files(x86)\Microchip\MPLABX\v5.00\ gnuBins\GnuWin32\bin;'
'C:\Program Files(x86)\Microchip\MPLABX\v5.00\mplabplatform\mplabipe;'
'C:\Program Files(x86)\Microchip\MPLABX\v5.00\mplab_platform\mplab_ipe\modules\ext;'
'C:\Program Files(x86)\Microchip\MPLABX\v5.00\sys\java \jre1.8.0_181\bin;'
```

One has to be careful when setting the command path as it is easy to accidentally overwrite all path information. For example, if the `'%path%'` is not specified in the presented code, all other path information will get deleted. One way to ensure that path can be easily re-created is to keep a copy of the path content in a separate file. [42] Neglecting the length of `path` may result in system failure as everything above 1024 characters will be deleted. To set the `path` safely, following steps needs to be made (applies to Windows systems):

⇒ Right click the Computer icon and choose Properties.

⇒ Click the Advanced system settings.

⇒ Click Advanced ⇒ Environment Variables. In the section System Variables, find the "Path" environment variable and edit it. If the "Path" environment variable does not exist, new one has to be made.

⇒ In the Edit System Variable (or New System Variable) window, add the value of the `added_path` as indicated above.

4.3.1 Building the project

Once the environment variables are added, the project can be built. In this section will be described, how the MPLAB project for a Double drive control was build by Windows command prompt, using MATLAB commands.

First, the working folder in MATLAB has to be changed to the same folder, where MPLAB project is saved. In this case:

```
% changing the working folder
project_path = strcat(current_folder,'\',project_name)
cd(project_path)
```

where `'project_path'` consists of the path to the MPLAB project. Then, the `make clean` command is used to delete files made during the last build. The command prompt is used from MATLAB by the `system` function:

```
% deleting old files
[status,cmdout] = system('make clean')
```

To build the default configuration of MPLAB project a command `make` was used.

```
% build
[status,cmdout] = system('make')
```

For more information about `make` see [43].

4.3.2 Uploading Firmware

After successful build, the `.hex` file can be uploaded. To do so, a command `ipecmd` was used as follows:

```
%% uploading the hex file
system_command = strcat('ipecmd -TPPK3 -P33FJ128MC804 -M ...
    -F',hex_path,' -OL ')
[status,cmdout] = system(system_command)
```

where these parameters have the following purpose:

- `TPPK3` is used to select specific programmer. This is, however, only one of two possible options. This option uses a "short name"Path" `PK3`, which stands for PICKIT 3. For PICKIT 4, `TPPK4` could be used instead. Second option is to use the serial number of used programmer, in the case of programmer used for this example it would be as `TSBUR121093960` for a serial number `BUR121093960`.
- `P33FJ128MC804` defines the used microcontroller. In this case the dspic33FJ128MC804 was used.
- `M` sets the program device memory regions. If used without any following parameter, as in this case, the whole memory will be programmed.
- `F` precedes a path to the `.hex` file, which is to be uploaded.
- `hex_path` contains full path to the `.hex` file.
- `OL` sets the release from reset.

More information and command line options can be also found in release notes for IPE command line interface [44] located also at[5]:

`C:\Program Files (x86)\Microchip\MPLABX\v5.00\docs\ReadmeforIPECMD.htm`

[5]Note that this link applies for MPLAB version 5.00.

5 Results and Discussion

A majority of this thesis has been about generating C codes from MATLAB and the results from that part will be discussed in this Chapter. However, it was also important to test that the generated files work properly.

In the first Chapter of this thesis, a list of functions which were to be tested for conversion was provided. All the functions and resulting C codes along with code generation reports in `.html` can be found in the attachment described in Appendix A. Not all of them will be commented in detail, because no issues occurred, but still, the following topics will be discussed:

- Description of testing and visualization of ADC filter.
- Generated code for finding eigenvalues
- Generating classes

Apart from these, a summary of code generation for Double Drive control will be discussed. In particular, the graphical user interface (GUI) will be described. The GUI was made in a way to link all the created tools together. This provided easy usage as well as it demonstrated all the steps taken for controlling Double Drive solely from MATLAB.

It is also well worth noting, that during the process of applying generated files, an error was encountered several times. It was related to a missing file `tmwtypes.h` which was used (called) in the generated files, but was not included in the "codegen" folder. It was found, that this file is located in:

```
C:\ProgramFiles\MATLAB\R2017b\extern\include\tmwtypes.h
```

and in case it is missing in the project it has to be copied there manually. The path above applies to MATLAB version R2017b.

Also, during the process of converting MATLAB functions to C, several minor findings were made. For example there is a difference in functionality of generated code for an inverse of a `3x3` matrix and for a matrix of a higher order. For the `3x3` matrix, the generated code does not contain any loops whereas the code for bigger matrices include several loops making the code more efficient.

In addition, the generated code might sometimes make use of already existing C programs. This was the case when converting MATLAB code for generating random numbers. The resulting code contained an already existing functionality with a copyright notice[1].

[1] Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura, All rights reserved.

5.1 Double drive control

An overview and the final design of Double Drive control is illustrated in Fig. 5.1. Although it shows a slightly simplified version of the actual structure, it depicts most of the processes and steps that were taken. The Figure can be read from top left, i.e., all the processes are initiated from MATLAB GUI. There, the PWM period can be set (if none is specified, a default value is used). Once the PWM period is set, the MATLAB functions for the C code generation can be converted. If there are later changes only to the PWM, there is also an option to generate files for PWM period only and to use the previously generated files for the rest. This was done only in the interest of saving time since generating only one function takes just a few seconds.

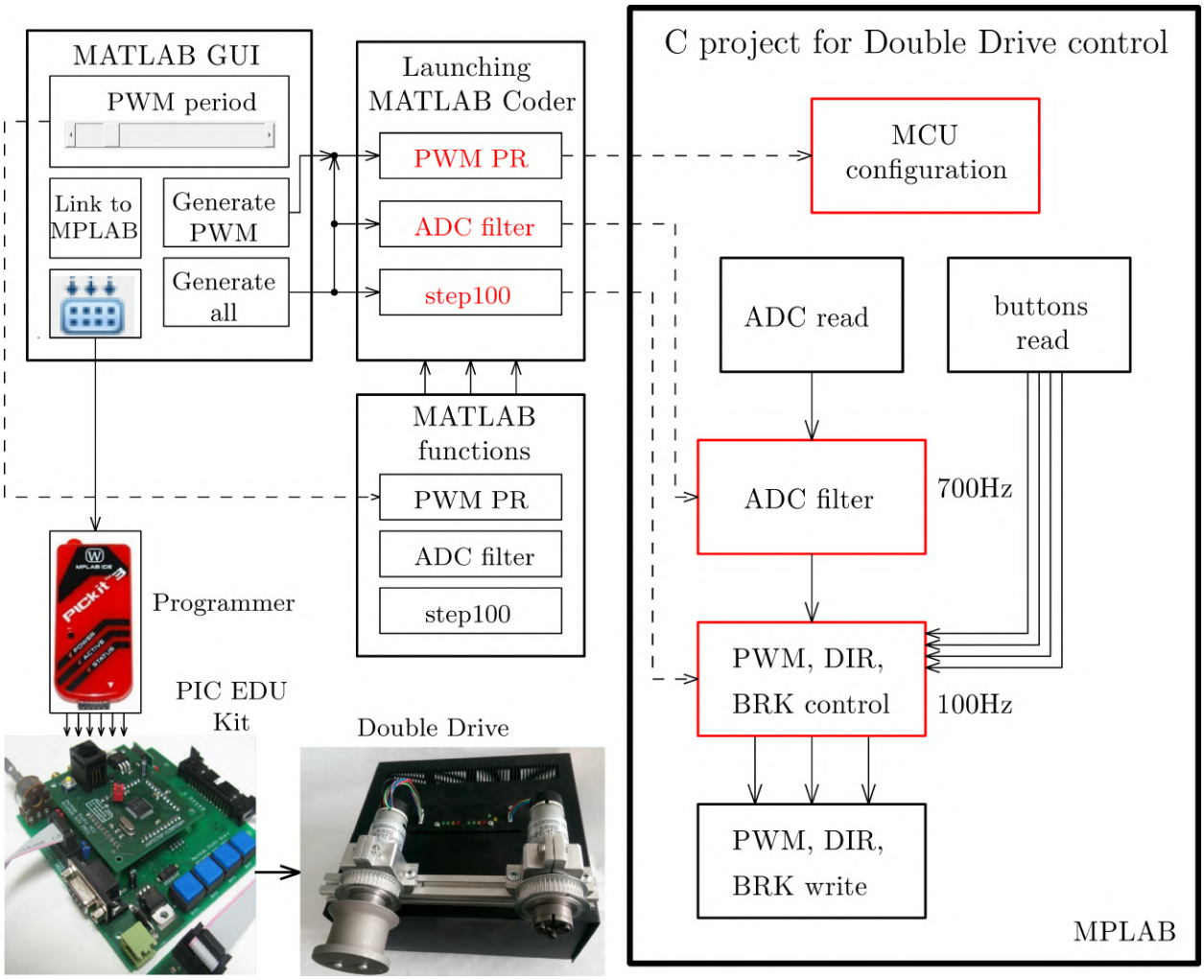

Figure 5.1: Utilizing MATLAB Coder to a fully automatic Double Drive control.

The "Generate all/PWM" option executes individual scripts (in Fig. 5.1 in red letters) which use the method of converting MATLAB code to C code described in section 4.2 "Code generation using command line". Those scripts make use of the MATLAB functions, which are in separate files. The functionality in those functions can be changed as long as the inputs/outputs stay the same. Also note, that such a direct modification is made, e.g. by the slider which changes the value of PWM period from the GUI (illustrated by a dashed line).

Once that all the files (`.c` and header files) are generated, they have to be copied into the MPLAB project. The button "Link to MPLAB" copies all the generated files to the MPLAB project folder. This step overwrites the previous files in that directory and has to be made after any changes (e.g. changing and generating the PWM period).

Subsequently, the MPLAB project can be built and flashed to the MCU in PIC EDU Kit using the PICkit 3 programmer. The process of building and flashing MPLAB projects from MATLAB has already been described in Section 4.3. After this step, the Double Drive should be working accordingly to the configured functionality.

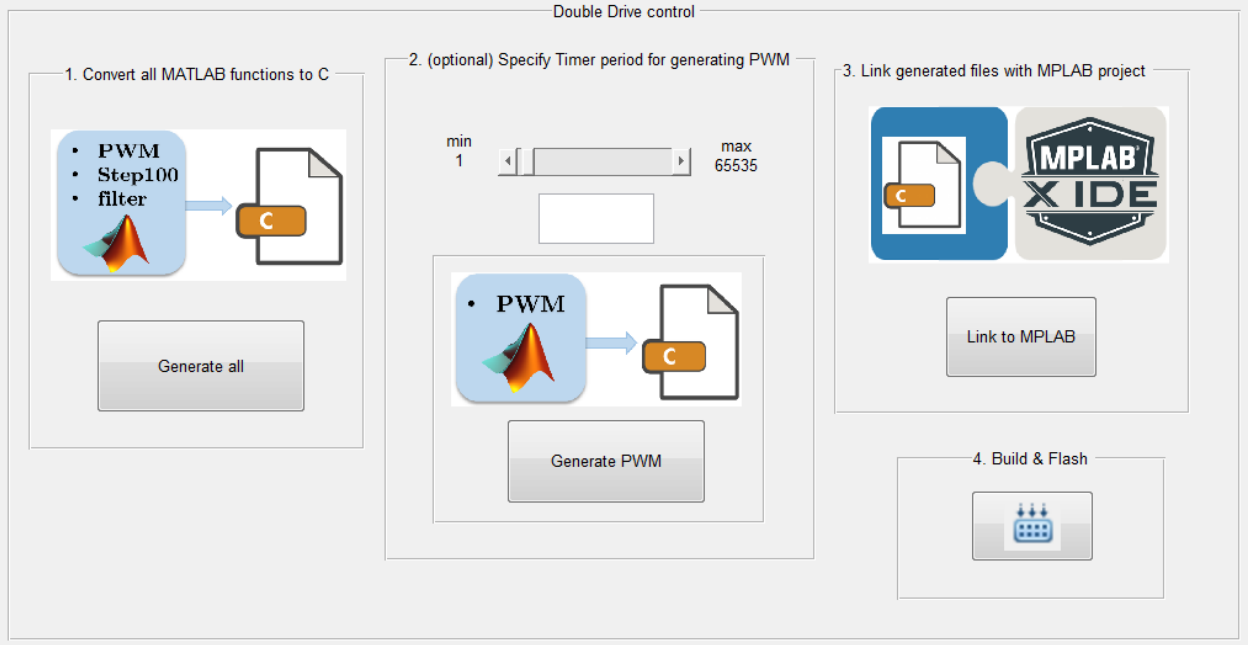

Figure 5.2: Graphical user interface enabling generation of configurable C code from MATLAB, linking with C environment, building and flashing the project to MCU.

The actual appearance of MATLAB GUI is shown in Fig. 5.2. It works in the same way as described in the text above. The only extra feature is that the PWM period can be set directly by typing a number. As a 16-bit timer was used for generating the PWM, the maximum value for period is 65535. If a bigger number is inputted, a warning is displayed and the value reduced to the specified maximum. The same principle applies for the inputs lower than 1.

5.2 Filtering Signals

In this section the process of testing the resulting C code for moving average filter will be described. MATLAB Code for the used filter can be found as an attachment described in Appendix A. Even though the code for filtering data is quite simple, the testing part proved to be challenging. PIC Edu Kit was used for the testing and the programming part was done in Simulink. In similar fashion the testing could be applied also to other examples. The Simulink block diagram is demonstrated in Fig. 5.3.

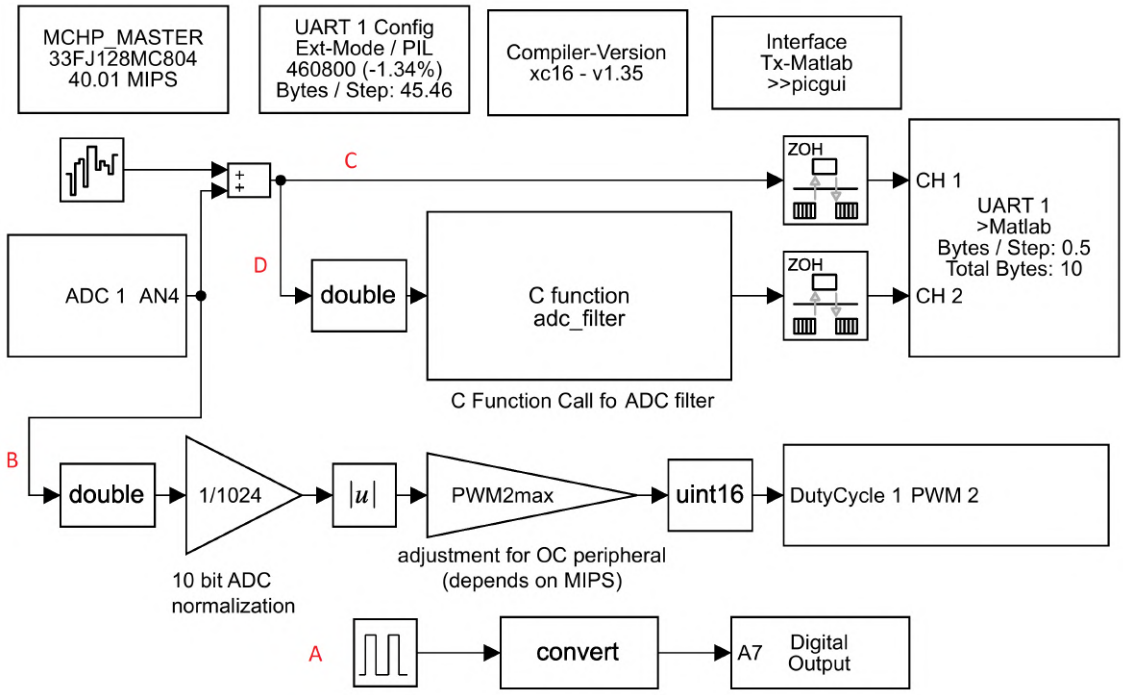

Figure 5.3: Simulink block diagram for testing generated C files by filtering data.

Four letters A, B, C and D can be seen by each signal in Fig. 5.3, all of which mark certain functionality described below:

- **A** sends a PWM signal with a 50% duty cycle and period of 2 seconds to a digital output (blinking LED). This is only used for signalization, that the program is running.
- **B** represents the same functionality as was used for controlling the speed of Double Drive motor. Only this time the signal was sent to another LED and was changing the brightness according to the 10 bit ADC signal.
- **C** marks ADC signal added with additional noise which is sent through the UART to receive and plot the data.
- **D** is analogous to C, however, this time the noisy signal is filtered by using a C function generated from MATLAB. A block "C function call" was used for this purpose.

The configuration of "C function call" for the filter function is illustrated in Fig. 5.4. As can be seen, the function takes and return a `double` value, that is because it has internal memory which is then averaging the consecutive inputs. The path to the `.c` file/s then have to be specified in the Simulink configuration window (likewise as in Fig. 5.8).

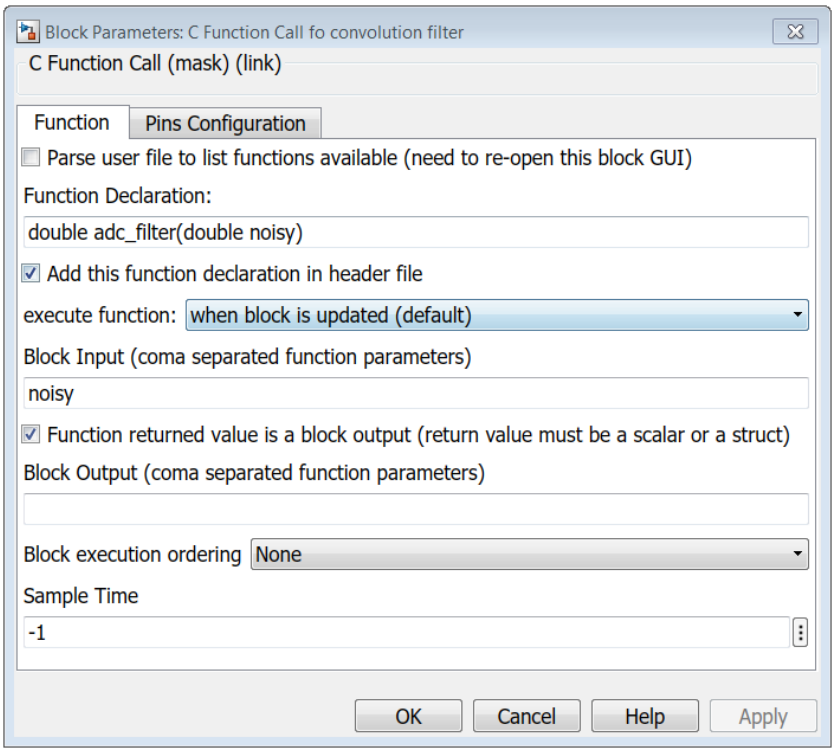

Figure 5.4: Configuration of C function call in Simulink for function `adc_filter.c`.

To display both the unfiltered (signal **C**) and filtered (signal **D**) data, the picqui interface was used (see Fig. 5.5). The baud rate was set to 460800 Bd.

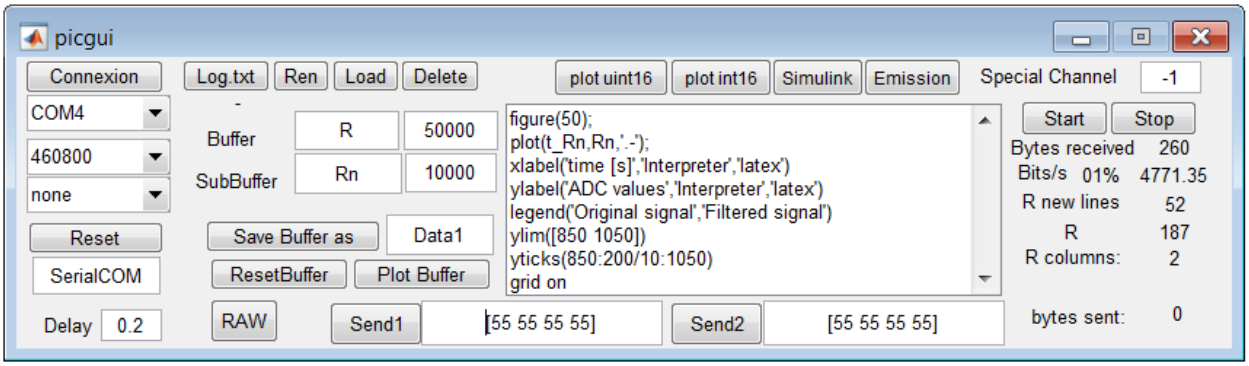

Figure 5.5: Configuration for picqui interface.

Finally, the signals are plotted in Fig. 5.6. As can be seen, the filter is working and the noise in the original signal is reduced. The filtered signal is not perfectly smooth, but the amount of filtering was not the main priority. The focus was mainly on the code generation itself together with creating a way to test the generated functions.

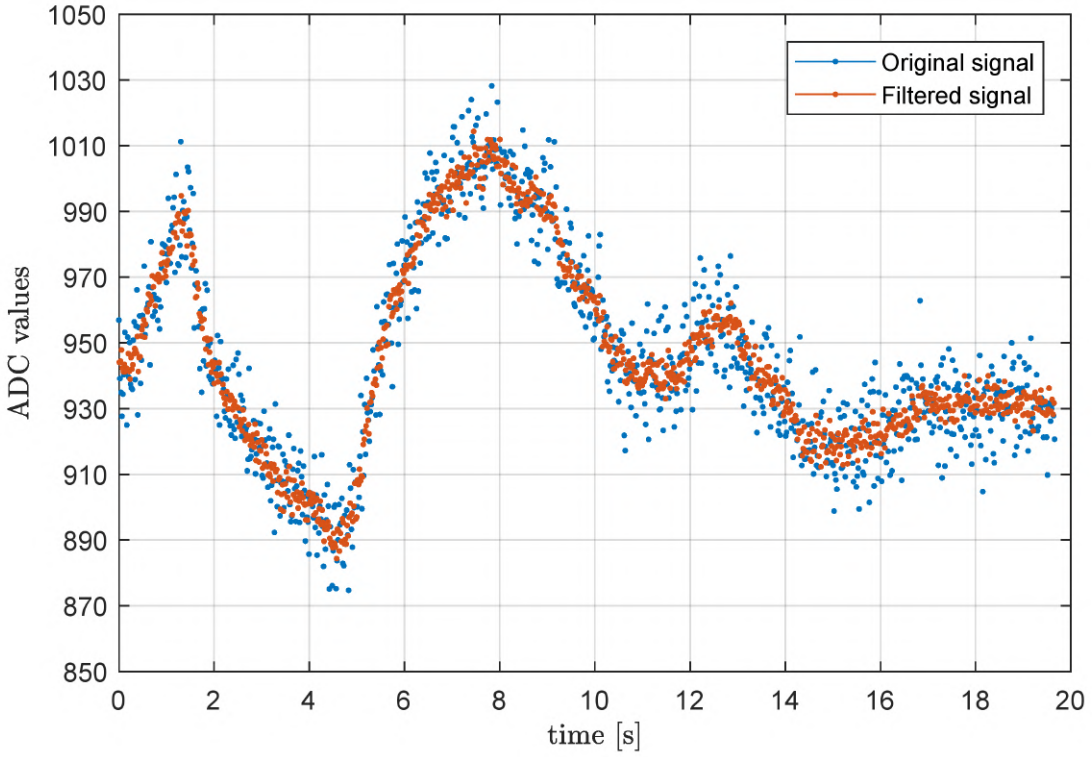

Figure 5.6: Testing of generated codes for filtering data.

5.3 Finding Eigenvalues

The complexity of tested functions for conversion was compared[2] and the generated codes for finding eigenvalues of a matrix was found to be one the most complex. Not only it contained around 2200 lines of code, 17 header files and 15 `.c` files in total, but also the interpretations of zeros was done in an inconvenient way (e.g. numbers like 2.22E-308 were used).

The difference in complexity between determining eigenvalues of a 3x3 matrix by using MATLAB and C is illustrated in Fig. 5.7. Of course anyone skilled in C language might be able to program it in a more efficient way, however, it would probably still take more time than to automatically generate functional code from MATLAB. Not to mention that it is very easy to change the size of matrix and to generate a new code for, e.g. matrix 100x100. This is where the huge potential of MATLAB Coder really shows up.

[2] According to the number of generated files as well as the total lines of code.

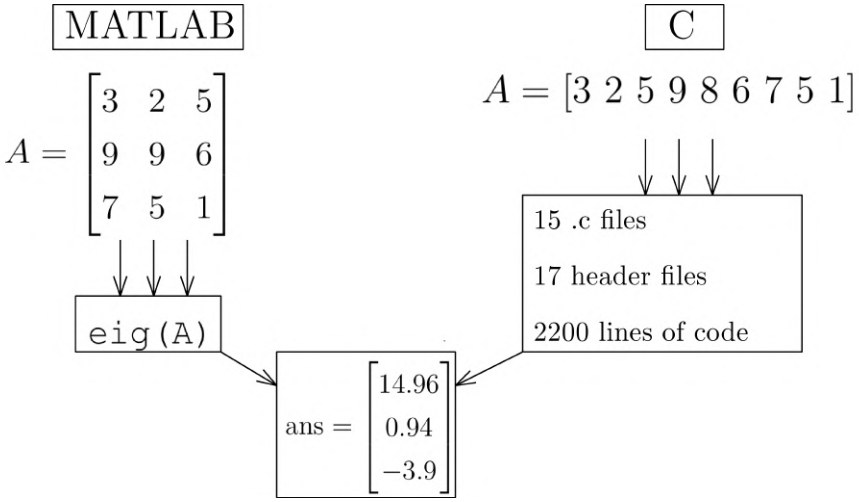

Figure 5.7: Comparison of determining eigenvalues of a 3x3 matrix by using MATLAB and generated C code.

The testing of generated code was carried out in a very similar way to the testing of filter in previous section. Most of the Simulink blocks were reused, only this time, the input to the C function block was an array of numbers (as shown in Fig. 5.7). Also all the locations of all the `.c` files had to be specified in model configuration as demonstrated in Fig. 5.8.

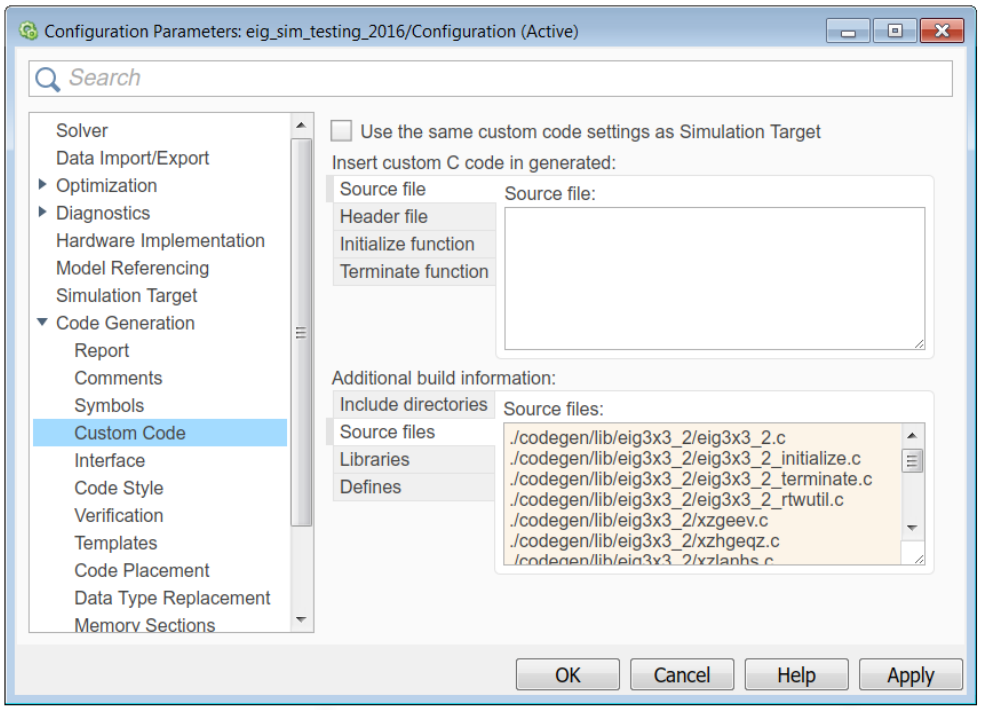

Figure 5.8: Model configuration and adding custom C code to Simulink for calculating eigenvalues of a 3x3 matrix by generated C code.

In spite the seeming complexity, the code worked very well and the results for calculating eigenvalues were correct. On the other hand, it would be very difficult to modify the functionality of already generated C codes. This is mainly because of the number of generated files as well as the fact that the naming of those files was quite random (Some of the names can be seen in Fig. 5.8, e.g. `xzgeev.c, xzhgeqz.c,`...)

5.4 Generating Objects

MATLAB enables the use object-oriented techniques, which can simplify programming some tasks that, e.g., involve specialized data structures, numerous functions etc. To further research MATLAB Coder possibilities, the generation of classes was tested. However, it was found, that the support for generating classes is not very broad. Even though it is possible to generate code for entry-point MATLAB functions that use classes, it is not possible to generate code directly for a MATLAB class. [45]

For example, if `Shape` is a class definition, it is not possible to generate code by executing something in the form of:

```
codegen Shape
```

Still, it is possible to generate code for functions that use classes. Nevertheless, the generated code will not contain any of the class functionality. An example of calculating area and distance of different shapes (individual shapes were represented by classes) was made[3]. A function was made which used mentioned classes and was therefore supposed to return values of calculated areas:

```
1 function [TotalArea, Distance] = use_shape()
2     s = Square(3,1,2); % Square(side,centerX,centerY)
3     r = Rhombus(3,4,7,10); % Rhombus(diag1,diag2,centerX,centerY)
4     TotalArea  = s.area + r.area; % answer is 15
5     Distance = Shape.distanceBetweenShapes(s,r);  % answer is 10
```

However, after successfully generating code for this function, the classes were ignored and the C function was in the following form:

```
void use_shape(double *TotalArea, double *Distance)
  {     *TotalArea = 15.0;
        *Distance = 10.0; }
```

It can be seen, that no functionality is included and the function only contains the resulting values which were corresponding to the initial input values (the same were obtained in MATLAB). If some functions for calculating the `TotalArea` and `Distance` were used, the generated code would contain this functionality as well. As such, the generation of functions from MATLAB that use classes was found unavailing.

[3] All of the codes and a `.html` code generation report is included in the attachment.

6 Conclusion

The aim of this diploma thesis was to explore the possibilities of generating C code from MATLAB by using the MATLAB Coder package. For this purpose, a simple application of controlling DC motor was created and several separate functions were tested. For the DC motor control, a C environment with e.g. MCU configuration was programmed. The control part was made in several MATLAB functions as well as the functionality to change the period of PWM. Some later changes to the functionality were made feasible (as long as the inputs/outputs to the function remained the same) and the function was then converted to C language by MATLAB Coder App. After code generation, the generated C files were linked with the C environment. In these conditions, the MATLAB Coder App has been used in two different ways.

First, the graphical interface of the App was utilized, which turned out to be a very good demonstration for the individual steps that have to be taken for code generation. Its main advantage has been that all the steps were controlled by the App and it was therefore easy to "catch" and fix any errors. However, it was quite laborious to perform modifications in setting or functionality of already generated functions. Also all of the functions had to be generated separately, which was not very practical.

Second, the command line was used for code generation. At first, it proved a bit challenging as it was not as straightforward as the MATLAB Coder App, however, once this technique was mastered, it became very effective. Not only was the use of command line faster to work with, it was also possible to reuse most the the code for several functions. Moreover, its potential also lied in the possibility of integrating scripts for code generation with another functionality in MATLAB. This allowed for creating a graphical user interface (in MATLAB), which then enabled, that the whole process of code generation, building, flashing to MCU and even making adjustments to the resulting C code functionality e.g. the PWM frequency, could have been easily performed.

In response to insufficient documentation of the use outlined above, a description of individual steps has been provided. It should allow anyone willing to use the MATLAB Coder for a similar purpose to be able to reuse the commands used in the practical part and therefore progress quickly in his/her work as well as to benefit from the referred bibliography.

The future work could focus on improving the MATLAB GUI for created tool and on extending the existing functionality. Also more complex functions could be integrated into the code generation. Another line of research worth pursuing further is to study the Embedded Coder extension which was mentioned several times in this thesis and which builds on MATLAB Coder and focuses more on reducing the memory usage or maximizing speed.

Summary of the Thesis' Outcomes:

- The C code generation for embedded applications has been inspected, mainly the use of MATLAB Coder but also the Blockset for Microchip dsPIC Microcontrollers, Simulink Support Package for Arduino Hardware or Embedded Coder were examined.
- An application in C language for DC motor control has been created which contained the configuration of selected peripherals (e.g. digital inputs/outputs, PWM, ADC) and a specific timing for calling automatically generated functions from MATLAB.
- Several MATLAB functions were converted to C language. These functions has been automatically linked with the created application.
- A MATLAB GUI has been made to allow the configuration of PWM period and also to simplify the use of the created tool. Use of the GUI allowed the PWM configuration, conversion of MATLAB code to C, linking the generated code with C application and a consecutive building and flashing.
- Several demonstrative tasks has been created to show the properties and also to verify the limits of MATLAB Coder. The main focus was on controlling a DC motor which included, e.g., ADC filtering and creating a functional code. However, also the limits for, e.g. generating objects were indicated.
- A detailed description of individual steps for code generations has been provided.

All of the master's thesis goals has therefore been fulfilled.

Bibliography

[1] Simulink Support Package for Arduino Hardware [online]. [cit. 2019-02-14]. Available: https://www.mathworks.com/help/supportpkg/arduino/index.html

[2] MPLAB® Device Blocks for Simulink® [online]. [cit. 2019-02-14]. Available: http://microchipdeveloper.com/simulink:start

[3] KERHUEL, Lubin. Lubin Kerhuel Website [online]. [cit. 2018-11-22]. Available: http://www.kerhuel.eu/wiki/Simulink_-_Embedded_Target_for_PIC

[4] ANDERSON, Rick and Dan CERVO. Pro Arduino: Arduino expert topics and techniques. USA: Apress Berkely, CA, ©2013. ISBN 1430239395 9781430239390.

[5] MATLAB Coder: User's Guide [online]. [cit. 2019-04-27]. Available: https://www.mathworks.com/help/pdf_doc/coder/coder_ug.pdf

[6] LYNCH, Kevin M., Nicholas MARCHUK and Matthew L. ELWIN. Embedded Computing and Mechatronics with the PIC32 Microcontroller. US: NEWNES, 2016. ISBN 978-0-12-420165-1.

[7] VEJLUPEK, Josef and Barnabás DOBOSSY. PIC EduKit User Guide. Mechlab, 2018. Brno University of Technology.

[8] MICROCHIP. Dspic33FJ128MC804 Datasheet [online]. Microchip Technology Incorporated. Printed in the U.S.A., 2012 [cit. 2018-11-25]. ISBN 978-1-62076-236-3. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/70291G.pdf

[9] Potenciometr PC1622NK010 [online]. [cit. 2019-05-04]. Available: https://www.gme.cz/pc1622nk010

[10] Potentiometer Construction, Working and Applications [online]. [cit. 2019-05-04]. Available: https://www.elprocus.com/potentiometer-construction-working-and-applications/

[11] Potentiometer Pin Diagram. In: Components101 [online]. [cit. 2019-05-01]. Available: https://components101.com/potentiometer

[12] VERLE, Milan. PIC mikrokontroleri [online]. Mikro Elektronika [cit. 2019-05-01]. Available: https://www.mikroe.com/ebooks/pic-microcontrollers-programming-in-assembly/introduction

[13] Pickit 3 supported devices [online]. [cit. 2019-05-02]. Available: https://pic-microcontroller.com/pickit-3-supported-devices/

[14] MPLAB® IDE USER'S GUIDE [online]. Microchip Technology [cit. 2019-05-02]. Available: http://ww1.microchip.com/downloads/en/devicedoc/51519a.pdf

[15] Microchip PICKIT3 Programmer Debugger [online]. In: . [cit. 2019-05-02]. Available: https://www.indiamart.com/proddetail/microchip-pickit3-programmer--debugger-2710959988.html

[16] PICkit™ 3 Programmer/Debugger User's Guide [online]. MICROCHIP [cit. 2019-05-02]. Available: https://modtronicsaustralia.com/wp-content/uploads/2013/02/PICkit_3_User_Guide_51795A.pdf

[17] PD3046: PD Series complete catalogue. Transmotec [online]. [cit. 2019-05-02]. Available: https://www.transmotec.com/search?s=PD3046

[18] Real-Time Toolbox and MechLab DoubleDrive Quick Start: mechlab.note 006. MechLab, 2012.

[19] LMD18201: H-Bridge [online]. Texas Instruments [cit. 2019-05-02]. Available: http://www.ti.com/lit/ds/symlink/lmd18201.pdf

[20] H-Bridges: The Basics [online]. [cit. 2019-05-02]. Available: http://www.modularcircuits.com/blog/articles/h-dge-secrets/h-bridges-the-basics/

[21] DROZDENKO, Benjamin, Ram SUBRAMANIAN, Kaushik CHOWDHURY a Miriam LEESER. Implementing a MATLAB-based Self-Configurable Software Defined Radio Transceiver. Department of Electrical and Computer Engineering, Northeastern University, 360 Huntington Ave, Boston, MA 02115.

[22] MATLAB Coder: Generate C and C++ code from MATLAB code [online]. [cit. 2019-02-14]. Available: https://www.mathworks.com/products/matlab-coder.html

[23] Using MATLAB with Other Programming Languages [online]. [cit. 2019-05-08]. Available: https://www.mathworks.com/products/matlab/matlab-and-other-programming-languages.html

[24] CHOU, Bill. MATLAB to C Made Easy [online]. [cit. 2019-05-08]. Available: https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/-campaigns/portals/files/matlab-coder/matlab-to-c-made-easy-presentation.pdf

[25] MOLER, Cleve. A Brief History of MATLAB. MathWorks [online]. [cit. 2019-05-08]. Available: https://www.mathworks.com/company/newsletters/articles/a-brief--history-of-matlab.html

[26] Founders: Cleve Moler, Chief Mathematician [online]. [cit. 2019-05-09]. Available: https://www.mathworks.com/company/aboutus/founders/clevemoler.html

[27] ANANTHAN, Arvind. Generating C Code from Your MATLAB Algorithms: A Brief History of MATLAB to C [online]. In: . [cit. 2019-04-24]. Available: https://blogs.mathworks.com/loren/2011/11/14/generating-c-code-from-your--matlab-algorithms/

[28] Using Embedded MATLAB Blocks. Goddard Consulting [online]. [cit. 2019-04-24]. Available: http://www.goddardconsulting.ca/simulink-using-embedded-matlab.html

[29] Generate C and C++ code optimized for embedded systems [online]. [cit. 2019-05-08]. Available: https://www.mathworks.com/products/embedded-coder.html

[30] ROULEAU, Guy. Welcome to the Coders! [online]. [cit. 2019-05-08]. Available: https://blogs.mathworks.com/simulink/2011/04/08/welcome-to-the-coders/

[31] Differences Between Generated Code and MATLAB Code [online]. [cit. 2019-05-04]. Available: https://www.mathworks.com/help/simulink/ug/expected-differences-in--behavior-after-compiling-your-matlab-code.html#bshcz7y

[32] Define Variable-Size Data for Code Generation [online]. [cit. 2019-05-08]. Available: https://www.mathworks.com/help/fixedpoint/ug/defining-variable-size-data--for-code-generation.html

[33] Supported Variable Types [online]. [cit. 2019-05-04]. Available: https://www.mathworks.com/help/simulink/ug/supported-variable-types.html

[34] Fixed-Point Data Types [online]. [cit. 2019-05-04]. Available: https://www.mathworks.com/help/fixedpoint/ug/fixed-point-data-types.html

[35] Fixed-Point Code Generation [online]. [cit. 2019-05-04]. Available: https://www.mathworks.com/help/fixedpoint/generate-and-deploy-fixed-point--code.html

[36] DEITEL, Paul J. a Harvey M. DEITEL. C: how to program. 6th ed. Upper Saddle River, N.J.: Pearson/Prentice Hall, c2010. How to program series. ISBN 01-361-2356-2.

[37] Alphabetical List of Functions and Objects Supported for C/C++ Code Generation [online]. Mathworks [cit. 2019-03-31]. Available: https://www.mathworks.com/help/coder/ug/functions-supported-for-code-generation-alphabetical-list.html

[38] Category List of Functions and Objects Supported for C/C++ Code Generation [online]. Mathworks [cit. 2019-03-31]. Available: https://www.mathworks.com/help/coder/ug/functions-supported-for-code-generation-categorical-list.html

[39] Convert MATLAB Code to Fixed-Point C Code [online]. [cit. 2019-04-04]. Available: https://www.mathworks.com/help/coder/ug/convert-matlab-code-to-fixed-point-c--code.html

[40] BALASUBRAMANIAN, L. (2018). Improving maintainability of time critical software by applying MATLAB Coder and architectural redesign : a case study in the PARIS domain Eindhoven: Technische Universiteit Eindhoven

[41] System Commands: Operating System Commands. Mathworks [online]. [cit. 2019-04-16]. Available: https://www.mathworks.com/help/matlab/ref/system.html

[42] STANEK, William. The Personal Trainer: Windows Command Line. United States of America: Stanek & Associates PO Box 362 East Olympia, WA 98540-0362, 2015.

[43] Building a Project Outside of MPLAB® X IDE. © 2019 Microchip Technology, Inc. [online]. [cit. 2019-04-25]. Available: http://microchipdeveloper.com/mplabx:work--outside-build-project

[44] Release Notes for IPE Command Line Interface [online]. [cit. 2019-04-25]. Available: http://denethor.wlu.ca/cp316/software/Readme_for_IPECMD.txt

[45] MATLAB Classes Definition for Code Generation [online], [cit. 2019-05-10]. Available: https://www.mathworks.com/help/simulink/ug/how-working-with--matlab-classes-is-different-for-code-generation.html

List of used symbols and abbreviations

DC	Direct Current
GND	Ground
I/O	I/O pin can be both Input and an Output
IDE	Integrated Development Environment
MATLAB	Matrix Laboratory
MCU	Microcontroller Unit
MEX	MATLAB Executable
MIPS	Million Instructions Per Second
PIC	Peripheral Interface Controller
PPS	Peripheral Pin Select
PWM	Pulse Width modulation
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus

List of figures

1.1 Blocks for programming dspic microcontroller - blinking LED. 21
1.2 Blocks for programming Arduino. 22
1.3 Initial plan for linking MATLAB Coder with MPLAB project for a specific example of Double drive control. 24
2.1 PIC EduKit used for the Double Drive control. [7] 27
2.2 Schematic for the connection of buttons. 29
2.3 Schematic of potentiometer [11]. 30
2.4 Pin diagram for dspic33FJ128MC804 microcontroller [8]. 31
2.5 Principle of Output Compare for generating a PWM signal. [8] 32
2.6 PICkit 3 Programmer/Debugger [15]. 32
2.7 Arduino UNO used for code generation from Simulink. 33
2.8 Configuration of Simulink for the use of Arduino UNO. 33
2.9 Double Drive - a laboratory model consisting of two brushed DC motors. . 34
2.10 Functional Block Diagram of LMD18201 H-Bridge [19] 34
2.11 Sign/Magnitude PWM Control of LMD18201 H-Bridge [19] 35
3.1 Different ways of utilizing MATLAB in code generation. Modification of schemes in [23] and [24]. 37
3.2 MATLAB Function block in Simulink. 38
4.1 Initial page of MATLAB Coder App. 43
4.2 Setting up numeric conversion. 44
4.3 Warning for `abs(x)` when a single precision conversion is selected. 45
4.4 Defining input types. 45
4.5 Checking for run-time issues. 46
4.6 Settings and generating code. 47
4.7 Successful code generation. 48
4.8 Final workflow. 48
5.1 Utilizing MATLAB Coder to a fully automatic Double Drive control. . . . 56
5.2 Graphical user interface enabling Kaka+Tomo generation of configurable C code from MATLAB, linking with C environment, building and flashing the project to MCU. 57
5.3 Simulink block diagram for testing generated C files by filtering data. . . . 58
5.4 Configuration of C function call in Simulink for function `adc_filter.c`. . 59
5.5 Configuration for picqui interface. 59
5.6 Testing of generated codes for filtering data. 60
5.7 Comparison of determining eigenvalues of a 3x3 matrix by using MATLAB and generated C code. 61
5.8 Model configuration and adding custom C code to Simulink for calculating eigenvalues of a 3x3 matrix by generated C code. 61

List of tables

2.1 Buttons on PIC Edukit and their use in Double Drive control. 28
2.2 LED indications on PIC Edukit for Double Drive control. 29
3.1 Useful functions for C code generation from MATLAB, particularly when using the command line option and not the MATLAB Coder App. 39

Appendices

A Attached files

In this Appendix the location (folder names) of all the scripts, files, models and projects will be described. The attachment includes:

- **`01_double_drive_project`** This folder includes the main project for controlling the Double Drive. This comprise of the GUI, scripts for converting MATLAB functions to C, MPLAB project, scripts for linking generated files with MPLAB and scripts for building and flashing.
- **`02_tested functions`** - all the tested functions are included here.
- **`03_eig_simulink_test`** contains the Simulink model for testing the generated function for calculating eigenvalues of a matrix.
- **`04_filter_simulink_test`** contains the Simulink model for testing generated function for filtering ADC signal.
- **`05_OOP`** contains the testing files for generating MATLAB classes.
- **`06_arduino_codegen_test`** contains the testing files for Arduino and Simulink Support Package for MPLAB (the files are called as Kerhuel).
- **`2019_DP_Macha_Tomas_162202.pdf`** - the diploma thesis in PDF.